Ian Stewart e Robin Jones

Il linguaggio macchina dello Spectrum



Il piacere del computer

Il piacere del computer

Il piacere del computer

1 *Tom Rugg e Phil Feldman* 32 programmi con il PET
2 *Rich Didday* Intervista sul personal computer, hardware
3 *Tom Rugg e Phil Feldman* 32 programmi con l'Apple
4 *Ken Knecht* Microsoft Basic
5 *Paul M. Chirlian* Pascal
6 *Tom Rugg e Phil Feldman* 32 programmi con il TRS-80
7 *Rich Didday* Intervista sul personal computer, software
8 *Herbert D. Peckham* Imparate il Basic con il PET/CBM
9 *Karl Townsend e Merl Miller* Il personal computer come professione
10 *Karen Billings e David Moursund* Te ne intendi di computer?
11 *Thomas Dwyer e Margot Critchfield* Il Basic e il personal computer, uno: introduzione
12 *Don Inman e Kurt Inman* Imparate il linguaggio dell'Apple
13 *Thomas Dwyer e Margot Critchfield* Il Basic e il personal computer, due: applicazioni
14 *Luigi Pierro* Il manuale del CP/M
15 *Carlo Sintini* A scuola con il PET/CBM
16 *David Johnson-Davies* Il manuale dell'Atom
17 *David E. Schultz* Il libro del Commodore VIC 20
18 *Jim Huffman e Robert Bruce* Il "debug" nei personal computer
19 *John M. Nevison* Programmazione in Basic per l'uomo d'affari
20 *Mark Harrison* Imparate il Basic con lo ZX81
21 *Ronald W. Anderson* Dal Basic al Pascal
22 *Herbert D. Peckham* Imparate il Basic con il Texas TI 99/44
23 *Sergio Borsani* A scuola con il Texas TI 99/4A
24 *Jerry Willis e Deborrah Willis* Come usare il Commodore 64
25 *Mark Harrison* Imparate il Basic con lo Spectrum
26 *Carlo Sintini e Costantino Mustacchio* A scuola con il Commodore 64
27 *David A. Lien* Imparate il Basic con l'IBM PC
28 *Ken Tracton* Introduzione al Lisp
29 *Fabio Mavaracchio* Programmi in Basic per l'elettronica
30 *Ian Stewart e Robin Jones* Il linguaggio macchina dello Spectrum

Ian Stewart e Robin Jones

Il linguaggio macchina dello Spectrum

franco muzzio & c. editore

Titolo originale *Spectrum Machine Code*
Traduzione di Tullio Policastro

Prima edizione: ottobre 1984
ISBN 88-7021-266-1

Via Bonporti 36, 35141 Padova, tel. 049/661147-661873

Indice

Prefazione

Questo libro è un'introduzione al linguaggio macchina dello Z 80, concepita espressamente per lo ZX Spectrum. Si assume che abbiate una ragionevole conoscenza del Basic, ma non sappiate assolutamente niente del linguaggio macchina: con l'aiuto di semplici esempi e progetti vi condurremo a un punto in cui sarete capaci di scrivere vostri programmi in linguaggio macchina, farli eseguire a partire da un programma in Basic, salvarli (SAVE) su nastro, e ricaricarli (LOAD) entro la macchina.
I principi di una buona programmazione e del linguaggio macchina dello Z 80 sono gli stessi per *ogni* computer che impiega una CPU Z 80. È vero; ma è ovvio che le cose diventano più facili se si parte da una descrizione che è stata fatta su misura proprio per la macchina di cui siete in possesso. Così, per risparmiarvi la fatica di mettere i punti sulle "i" ed i tagli alle "t", ovvero di adattare i vari programmi, troverete qui il lavoro bello e fatto e pronto per l'uso.
Il vantaggio principale presentato dal linguaggio macchina è che può svolgere compiti diversi che sono *possibili* anche con l'ausilio del Basic, ma in maniera troppo lenta per essere accettabile. Lo svantaggio principale è che si pongono al programmatore esigenze maggiori, richiedendogli di tenere minuto conto, in tutti i dettagli, della collocazione esatta delle informazioni entro la macchina, e sotto quale forma, e del modo con cui lo Spectrum le interpreterà. In cambio, se imparate *davvero* il linguaggio macchina, verrete anche a conoscere molte cose riguardo la vostra macchina!
Cominceremo fissando alcuni concetti di "teoria": come vengono me-

morizzati nel computer i numeri, come vengono trattati i numeri negativi, e come operano i codici binari ed esadecimali (indispensabili in questo campo). Poi, studieremo l'architettura di una versione *semplificata* dello Z 80, esaminandone gli aspetti principali (registri, modi di indirizzamento, indicizzazione e in-direzionamento, il contatore di programma e lo *stack pointer* o puntatore della pila), senza entrare nei minuti dettagli. Se ci si riferisse direttamente allo Z 80, ogni concetto andrebbe precisato continuamente con dei "se", "ma", "però": questo microprocessore è un "animale" molto sofisticato, e risulta senz'altro molto più facile esaminarne il funzionamento confrontandolo con qualcosa di più semplice.
Poi verrà descritto lo Z 80 vero e proprio, e verrà esaminato nei dettagli un importante gruppo di istruzioni – le istruzioni LD. Ciò allo scopo di spiegare pure le differenze dei vari "modi di indirizzamento" nel linguaggio macchina.
Spiegheremo il modo di memorizzare, eseguire, salvare e ricaricare il linguaggio macchina, elaborando un programma Basic che renderà tali compiti di facile realizzazione. Tutti i successivi programmi descritti in questo libro utilizzeranno questo programma Basic.
Fra le varie "routine" discusse troviamo un programma per eseguire una moltiplicazione in linguaggio macchina (con esemplificazione di altre importanti istruzioni). Descriveremo nei particolari come viene controllato lo schermo, e come esso viene manipolato da parte dello Spectrum; capitoli distinti esamineranno utili routine in linguaggio macchina relative alla memoria degli attributi e alla memoria dello schermo (fra cui alcune per realizzare "scroll" di vario tipo, cambiamenti di colore, generazione di effetti ottici, commutazione ON/OFF del lampeggio).
Vengono poi descritti in maggior dettaglio i vari tipi di *flag*, basandosi sull'esempio di una routine di rinumerazione delle linee di un programma Basic. Vengono poi presentati due potenti gruppi di istruzioni, per la ricerca e il trasferimento di blocchi di contenuti di memoria, che poi vengono applicati ad altri vari modi di "scrolling". Un capitolo finale copre vari altri concetti non trattati prima.
Le appendici comprendono tavole di informazioni utili alla programmazione in linguaggio macchina: per la conversione esadecimale/decimale; per riservare zone di memoria per il linguaggio macchina; sulle variabili di sistema; sulle istruzioni dello Z 80 e sui loro effetti sui flag Zero e Carry; una lista dei codici operativi e del loro valore esadecimale (in ordine alfabetico); e infine un utile "assembler" parziale scritto in Basic (HELPA), che vi permette di scrivere, modificare e mandare in esecuzione un programma in linguaggio macchina senza eccessive difficoltà. In particolare, questo programma calcola automaticamente i valori dei salti relativi, risparmiandovi un sacco di lavoro noioso (e che quindi

tende a dare risultati erronei, con possibili conseguenze disastrose, quando viene svolto "a mano").
Il libro è scritto in una maniera che vi rende possibile usare con profitto, se lo desiderate, le varie routine in linguaggio macchina *senza* bisogno di capire come funzionano. Tuttavia speriamo che il vostro scopo sia più gratificante: ossia quello di imparare a scrivere *proprie* routine in linguaggio macchina.
Sin qui ci siamo riferiti a noi stessi con "noi": però, come è successo negli altri nostri libri, troviamo che la cosa finisce col non funzionare più bene andando avanti. Perciò, a partire da questo istante, ci riferiremo a noi come "io". E quando useremo (userò) "noi", si intenderà "io e il lettore". A prima vista può suonare un po' strano, ma credeteci, in questo modo finisce per essere più informativo.

CAPITOLO UNO

Per stuzzicare il vostro appetito...

Non avreste acquistato, o non stareste sfogliando questo libro, se non aveste sentito dire che il vostro Spectrum è capace di fare cose sorprendenti, e in modo assai veloce, usando qualcosa che si chiama *linguaggio macchina*. Ed è proprio così: però il guaio con il linguaggio macchina è che esso, a differenza del Basic, non pensa al posto vostro. Dovete prestare molta più attenzione ai dettagli più minuti, e tenere sempre d'occhio la collocazione del vostro linguaggio macchina nella memoria. Il linguaggio macchina *non* si può proprio dire "*user friendly*"; tanto per cominciare assomiglia alquanto ai geroglifici egiziani, e ha il fascino e la chiarezza di un elenco telefonico scritto in Urdu (dialetto africano...). Beh, a dire la verità, le cose non sono poi così gravi, e voi potrete presto capirne qualcosa: però sia chiaro che dovrete metterci dell'impegno, prima di ricavarne davvero un profitto. E perciò, per convincervi che dopotutto la cosa merita, incomincerò col presentarvi alcune "routine" in linguaggio macchina che generano sullo schermo a grande velocità degli effetti, di carattere un po' "astratto". Se siete in grado di fare la stessa cosa anche col Basic, allora certamente la risoluzione dei problemi dell'economia mondiale è proprio quello che siete capaci di fare a mente mentre state facendo colazione...
Per ora non cercate di capire come funziona: lo vedremo più avanti. Limitatevi a copiare il programma nella memoria del computer, ed a lanciarlo con RUN. Dovrete probabilmente usare alcuni tasti cui avete finora fatto ricorso di rado, come

USR (tasto "L" in modo "esteso" - cursore E)
CLEAR (tasto "X" in modo comandi - cursore K)

nonché

POKE (tasto "0" in modo comandi)

Eccovi il primo programma:

```
10 CLEAR 31999
20 BORDER 0
30 DATA 1, 0, 3, 17, 0, 88, 33, 0, 0, 237, 176, 201
40 FOR i = 0 TO 11
50 READ x
60 POKE 32000 + i,x
70 NEXT i
80 PAUSE 0
90 LET y = USR 32000
```

State attenti a impostarlo correttamente, in particolare per i valori entro la lista dei DATA. E ora date RUN. Lo schermo rimarrà inerte sino a quando non premete un tasto qualunque (a causa dell'istruzione di linea 80). Ora premete un tasto: avrete una risposta *istantanea*, con lo schermo che si è riempito di quadratini di diverso colore, alcuni dei quali lampeggianti. (Se non è così, controllate bene il listato. Potrebbe darsi che per poter fare questo dobbiate togliere la spina dell'alimentazione, e a questo punto dovrete ovviamente ricominciare tutto da capo, se non avrete avuto l'accortezza di "salvare" su nastro il programma, che potrete allora ricaricare e controllare. Questo è uno dei caratteristici "difetti" del linguaggio macchina!)



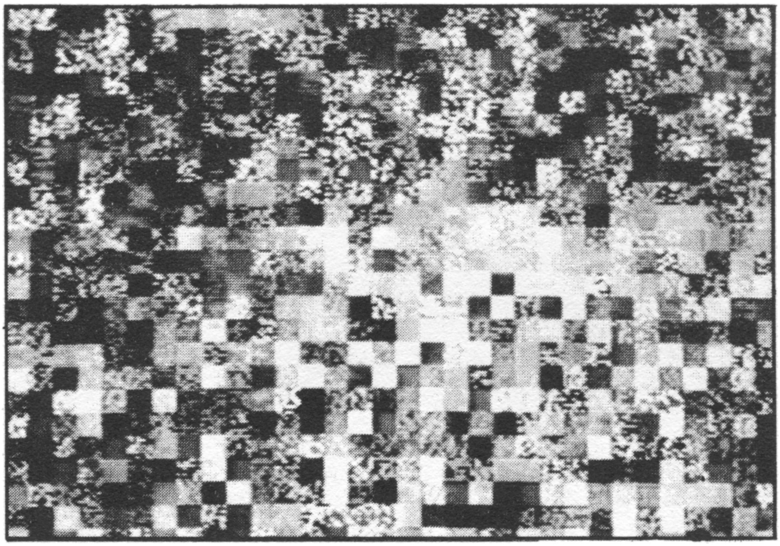


FIG. 1. "Scroll" laterale di un'infinità di quadratini, colorati casualmente, alcuni screziati, alcuni brillanti, altri lampeggianti...

Veloce, ma non drammatico. Il passo successivo consiste nell'apportare qualche piccolo cambiamento al programma. Cancellate (con 90 seguito da ENTER) la linea 90, e aggiungete le seguenti linee:

```
100  LET t = 0: LET s = 0
110  IF t > = 256 THEN LET t = t - 256 * INT (t/256):
     LET s = s + 1
120  POKE 32007, t: POKE 32008, s
130  LET y = USR 32000
140  LET t = t + 1
150  GO TO 110
```

Date RUN, e premete un tasto qualsiasi per avviare lo spettacolo. Otterrete uno schermo simile al precedente, ma in cui stavolta si osserva un movimento *laterale* a una ragionevole velocità di spostamento. Provate ora a modificare la linea 140 in

```
140 LET t = t + 32
```

e avrete uno spostamento verso l'alto; modificatela in

```
140 LET t = t + 31
```

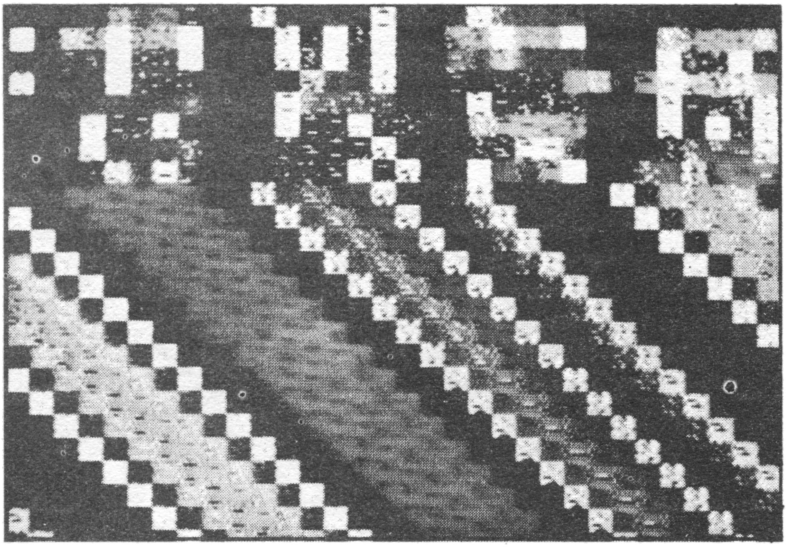


FIG. 1.2. ... che acquistano ora una certa struttura: configurazioni striate che ruotano velocemente...

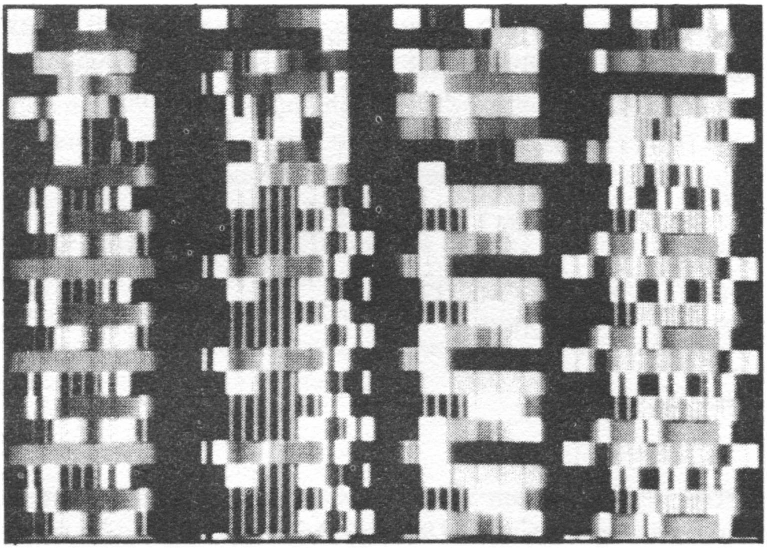


FIG. 1.3. ... questa però non sarete capaci di scorgerla! Le immagini scorrono ora troppo veloci...

e avrete uno "scroll" *diagonale*. Provate qualche altro valore da sommare a t nella stessa istruzione.

Ed ora, ritornate la 140 nella forma LET t = t + 1, e modificate invece l'istruzione DATA così:

30 DATA 1,0,24,17,0,64,33,0,0,237,176,201

e ripetete l'intera procedura. Non vi dirò quello che potrete vedere: è una sorpresa! Anche stavolta potete provare a modificare la linea 140 nei modi visti prima.

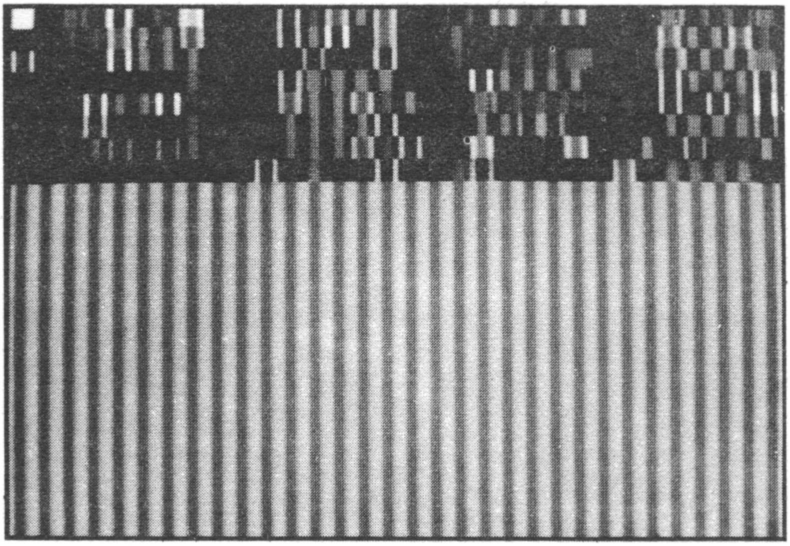


FIG. 1.4 ... ora una pausa, con gradevoli strisce in verde...

UNO SHOW "OTTICO"

Insolito, certo; ma non si può ancora dire spettacolare. E allora proviamo a combinare le due routine insieme, con qualche piccolo aggiustamento:

```
 10  CLEAR 31999
 20  BORDER 0
 30  DATA 1, 0, 24, 17, 0, 64, 33, 0, 0, 237, 176, 201
 35  DATA 1, 0, 3, 17, 0, 88, 33, 0, 0, 237, 176, 201
 40  FOR i = 0 TO 23
 50  READ x
 60  POKE 32000 + i, x
 70  NEXT i
100  LET i = 0: LET s = 60
110  IF t > = 256 THEN LET t = t - 256 * INT (t/256):
     LET s = s + 1
```

```
120  POKE 32007, t: POKE 32008, s
125  POKE 32019, t: POKE 32020, s - 1
130  LET y = USR 32000
140  LET y = USR 32012
150  LET t = t + 1
160  GO TO 110
```

Provate a dare RUN, e lasciate che vada avanti per un minuto o due: al principio l'andamento è abbastanza pacifico, ma gradualmente lo schermo sembra impazzire...
Provate a modificare la linea 150 con t + 32, t + 31, ecc., come prima.
Provate a modificare le inizializzazioni della riga 100. Cosa succede se iniziate con s = 0? o s = 40?

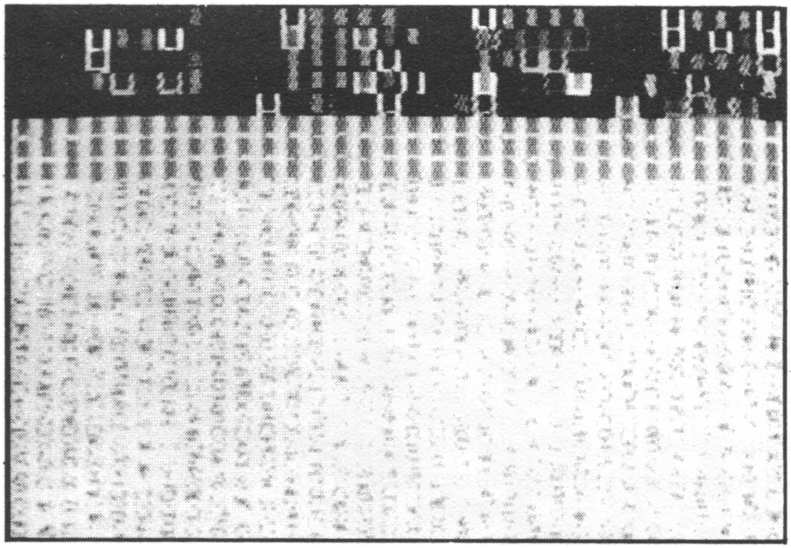


FIG. 1.5. ... ma eccoli che arrivano, come orde di formiche arrabbiate, che divorano tutto quello che incontrano sul loro cammino!

Se provate con s = 63, l'azione diventa così veloce da essere difficile da seguire. Se aggiungete la linea

```
145 IF INKEY$=""THEN PAUSE 0
```

potrete notare che tutto si ferma sino a che non premete un tasto, e che quindi così potete fare avanzare "per gradi" il programma in ogni

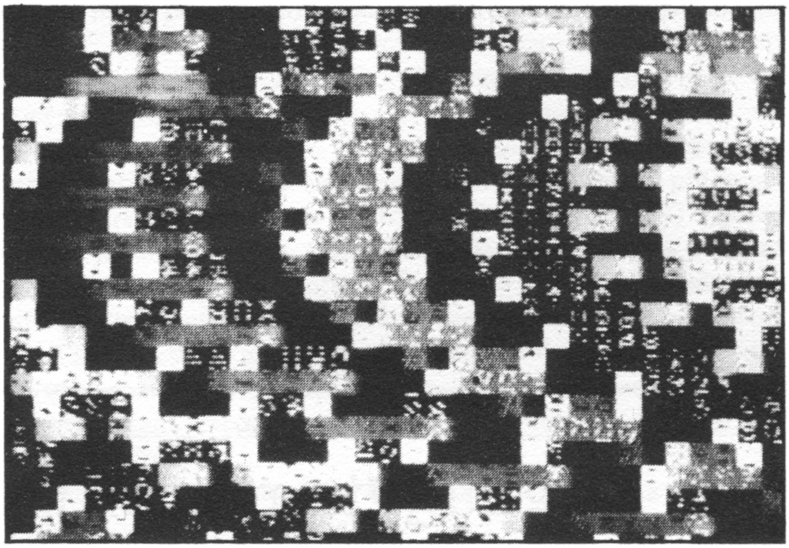


FIG. 1.6. ... per passare poi ad intense macchie colorate rotanti. E tutto a partire da un paio di dozzine di istruzioni in linguaggio macchina (basta una macchina con 16 K RAM). E questi sono solo degli esempi!

momento, semplicemente premendo un tasto e togliendo poi subito via il dito, e ripetendo la cosa più volte. Potrete così ammirare dei "quadri" che in precedenza passavano sullo schermo troppo rapidamente durante la folle avanzata.
Potete inventare una serie infinita di variazioni a questo programma: badate solo a non toccare le linee 30, 35, 130 e 140, dove sta il cuore del linguaggio macchina e della sua esecuzione.
Confido che a questo punto sarete convinti che il linguaggio macchina ha qualcosa in più da offrire. Tuttavia, a parte la generazione di simpatici effetti ottici variabili ad alta velocità, non si può proprio affermare che questo particolare esempio produca qualcosa di realmente *utile*. Il suo pregio sta nel fatto di essere costituito da uno spezzone relativamente corto di linguaggio macchina, che genera un effetto sproporzionato. Per fare buon uso di programmi in linguaggio macchina, dobbiamo essere capaci di scriverli in modo appropriato, e con uno scopo specifico (così come avviene per un buon programma in Basic). Il resto di questo libro si prefigge proprio questo scopo.

CAPITOLO DUE

I numeri in linguaggio macchina

Siamo abituati a pensare ai numeri in termini "decimali". Se scrivo il numero 3814, tutti comprendiamo che cosa significa:

$3 \times 1000 + 8 \times 100 + 1 \times 10 + 4 \times 1$

e possiamo vedere che per ricavare il "valore di posizione" di una cifra rispetto a quella alla sua destra dobbiamo moltiplicare il precedente valore di posizione per 10 (1, poi 10, poi 100, ...). Si dice che il dato numero è in *base* dieci.

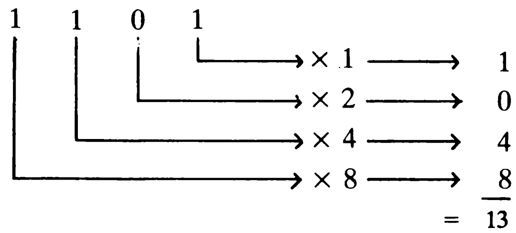
Dato che siamo stati abituati a fare così da tanto tempo, è difficile comprendere che esistono altri sistemi, perfettamente razionali, per fare lo stesso lavoro. I primi progettisti di calcolatori certo non fecero in questo modo: essi usarono nelle loro macchine di calcolo rappresentazioni di numeri in base dieci, e così andarono incontro ad alcuni inconvenienti davvero sgradevoli. Per lo più questi dipendevano dal fatto che gli amplificatori elettronici non hanno sempre lo stesso comportamento per tutti i segnali che vengono loro applicati in ingresso. Per esempio un amplificatore che si suppone fornisca in uscita un segnale "doppio" di quello in entrata, farà così per segnali di ampiezza pari a 1, 2, 3 e 4 unità in entrata; ma poi esso finisce per "saturarsi", cosicché per esempio, un segnale di ingresso pari a 5 produce un'uscita di solo 9.6, un segnale di 6 produce 10.8, e magari risulta assai difficile distinguere le due uscite per ingressi pari a 8 e 9 unità.

Inserite un nastro musicale nel vostro registratore, e aumentate il volume di uscita. Sentite la distorsione che c'è nelle parti più "forti"? È lo stesso effetto di prima.

I pionieri nella progettazione dei computer non avvertivano questa distorsione; semplicemente trovavano che le loro macchine non erano capaci di distinguere valori diversi, in certi casi, e questo era imperdonabile per un computer. Dovettero quindi "ripensare" il loro sistema di rappresentazione dei numeri, in modo da adattarsi nel miglior modo alle particolarità dei sistemi elettronici. La cosa realmente più semplice che si può fare con un segnale elettrico è di "inserirlo" (ON) o "disinserirlo" (OFF), così come si "accende" o si "spegne" una lampadina elettrica agendo su un interruttore. In questo modo è possibile rappresentare soddisfacentemente i numeri 0 (per esempio con lo stato OFF) e 1 (ON). La distorsione non svolge più alcun ruolo. Per quanto "distorto" sia un segnale, si può sempre riconoscere quando esso c'è o non c'è. Ma come possiamo escogitare un sistema di numerazione che usi solo 0 e 1 come sue cifre?

È possibile. In un sistema a base 10, il massimo numero a una cifra è 9. Aggiungiamo un'unità a 9 e otteniamo 10 – c'è stato un *riporto* alla cifra successiva. Possiamo scrivere qualsiasi numero usando una base qualsiasi, e il massimo numero rappresentabile con una singola cifra sarà sempre 1 meno della base. Se la base è 2, la massima cifra è 1, cosicché un sistema in base 2 (o *binario*) comprende solo le cifre 0 e 1.

Se abbiamo un numero binario come 1101, lo possiamo convertire nel corrispondente numero decimale (base 10) così:



Anche la conversione inversa è semplice: prendiamo per esempio 25. Se scriviamo i "valori di posizione" delle cifre in un numero binario

32 16 8 4 2 1

e operiamo a partire da sinistra, è evidente che ci serve un 16 (primo valore inferiore a 25), che ci dà un resto di 9, che a sua volta sarà formato dalla somma di 8 ed 1, cosicché 25 vale in binario

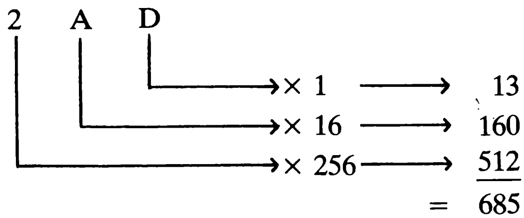
0 1 1 0 0 1 = 11001

I CODICI ESADECIMALI

La cosa funziona bene per valori relativamente piccoli, ma diventa un po' scomoda per valori grandi. Esistono vari metodi per la conversione rapida, e non è difficile reperire su qualche libro listati di programmi per la conversione binario/decimale e decimale/binario; ma desidero esaminare un procedimento che si serve del codice *esadecimale*, perché ci sarà utile in seguito.
Un numero esadecimale è un numero in base 16. Pertanto, i valori di posizione delle varie cifre si ottengono per successiva moltiplicazione per 16. I primi 5 valori sono:

16^4	16^3	16^2	16^1	16^0
65536	4096	256	16	1

"Un momento!" – state tutti dicendo – "Sono numeri complicati, e ad ogni modo abbiamo appena detto che la cifra massima può valere 15. Cosa succede?"
Abbiate fiducia! In primo luogo, risolveremo il problema delle cifre superiori a 9 assegnando i valori da 10 a 15 alle lettere da A a F. Perciò, il numero 2AD scritto in esadecimale si converte in decimale così:



(NB: D = 13; A = 10)

E ora accenniamo a una comoda particolarità del sistema esadecimale. Dato che 16 costituisce anche uno dei valori di posizione del sistema binario (il quinto), ne viene che ciascuna cifra esadecimale in un numero può essere sostituita dalle 4 cifre binarie che la rappresentano. (Ricordiamo che, per brevità, invece di "cifra binaria" si usa spesso dire "bit", che è l'abbreviazione del termine corrispondente inglese "*binary digit*".) La tabella che segue mostra questa conversione.

Decimale	Esadecimale	Binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Una tabella più estesa è fornita nell'appendice 1. *Nota*: In questo libro, scritto appositamente per lo Spectrum, le lettere le potrete trovare scritte in minuscolo o maiuscolo: ma negli **INPUT** si suppone che vengano sempre fornite lettere minuscole (senza premere **CAPS SHIFT**, per intenderci) per comodità.
Ora, supponiamo di voler convertire il numero (decimale) 9041 in esadecimale.
Dobbiamo prima togliergli i multipli di 4096, poi quelli di 256, e così via, come qui di seguito:

		9041
2 × 4096	=	8192 −
		849
3 × 256	=	768 −
		81
5 × 16	=	80 −
		11
1 × 1	=	1
		0

Quindi, la rappresentazione in esadecimale è 2351.
E ora semplicemente copiamo i codici binari delle cifre dalla tabella:

2	3	5	1
0010	0011	0101	0001

e otteniamo il corrispondente binario del decimale 9041: basta riunire assieme i 4 blocchi e abbiamo 0010001101010001.
La conversione da esadecimale a binario è tanto semplice che molto spesso si lasciano i numeri in esadecimale anche se in definitiva ci dobbiamo servire dei valori binari. Dopo tutto, è abbastanza facile commettere degli errori nel ricopiare una lunga fila di 0 e 1.

CONVERSIONE TRAMITE COMPUTER

Eccovi un programma per la conversione da numero decimale a numero esadecimale. Non fa altro che dividere il numero per 16 e considerare il relativo resto: perciò le cifre vengono "estratte" nell'ordine inverso a quello mostrato prima.

```
  20>LET p=4
  30 LET h$="0000"
  40 INPUT "N.o decimale (max.65
535) ";nd: LET d=nd
  50 LET n=INT (nd/16)
  60 LET r=nd-16*n
  70 LET h$(p)=CHR$ (r+48+39*(r>
9))
  80 LET nd=n
  90 LET p=p-1
 100 IF nd>0 THEN GO TO 50
 110 PRINT '"Il numero decimale
";d'"corrisponde in HEX a  ";h$
```

La linea 70 serve per ricavare le cifre e le lettere da a ad f (ricordiamo che abbiamo scelto le *minuscole*, che si prestano a minori confusioni, per esempio fra B e 8) da selezionare. A prima vista può sembrare un po' confusa, ma ciò deriva dal fatto che i codici ASCII impiegati dallo Spectrum per la rappresentazione interna dei caratteri si presentano un po' scomodi. Vogliamo contare in successione ..., 7, 8, 9, a, b, ..., e sarebbe comodo che il codice per a fosse immediatamente successivo a quello per il 9. Sfortunatamente, si trova invece a quaranta posti di distanza, vale a dire la differenza è troppo grande di 39. Dobbiamo sommare 39 per i caratteri che vanno oltre il 9. L'espressione logica r>9

ha il valore 1 se risulta vera, e 0 se falsa; il valore 39 viene sommato solo quando il carattere è fra a e f. (Il valore 48 inoltre deriva dal fatto che il codice ASCII per lo 0 è 48).
Il risultato viene in ogni caso fornito sotto forma di numero esadecimale di 4 cifre, con eventuali 0 anteposti non significativi, e con le lettere in minuscolo, perché le cifre esadecimali vengono da noi sempre scritte con lettere minuscole, e questo serve a ricordarcelo. Il programma non funziona se il risultato dovesse contenere più di 4 cifre esadecimali, ma per i nostri scopi va benissimo lo stesso, perché, a causa dei limiti di memoria dello Spectrum, 4 cifre sono sufficienti in ogni caso.
Ed ecco il programma per la conversione inversa (da esadecimale a decimale):

```
 140>INPUT "Imposta valore HEX (
4 cifre) "'h$
 150 LET nd=0
 160 FOR p=1 TO 4
 170 LET nd=nd*16+CODE h$(p)-48-
39*(h$(p)>"9")
 180 NEXT p
 190 PRINT "Il valore decimale d
i ";h$;"H"'"e'",nd
```

Anche qui troviamo una formula un po' complicata nella linea 170, che non è altro che l'inversa di quella della linea 70.
Potremmo assiemare le due routine e farle precedere da un piccolo menu come il seguente:

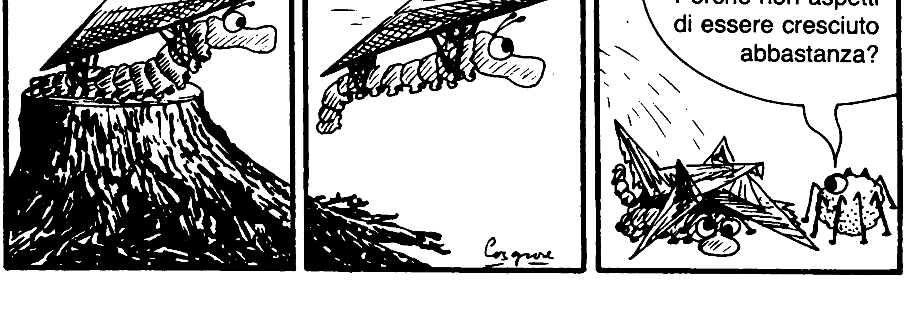
```
   2 CLS : PRINT "CONVERSIONE HE
X/DECIMALE"
   3 PRINT '"1)  DEC->HEX"''"2)
 HEX->DEC"''"3)  STOP"
   5 INPUT "SELEZIONE : ";sel
   8 IF sel=1 THEN GO SUB 20
   9 IF sel=2 THEN GO SUB 140
  10 IF sel=3 THEN STOP
  12 PAUSE 0: GO TO 2
```

Naturalmente a questo punto dovremo inserire dei RETURN come linee 120 e 200 addizionali.

CAPITOLO TRE

Positivo e negativo

Ora che abbiamo visto qualche concetto sulla manipolazione dei numeri binari, torniamo a considerare come essi vengono trattati internamente alla macchina. Solitamente, un numero è contenuto in un dato numero fisso di bit, comunemente 16, 24 o 32, a seconda del tipo di macchina. Questo numero di bit viene detto *dimensione di una parola* della macchina (o *lunghezza* di una parola).
Vediamo per esempio, nella tabella della pagina a fronte quali numeri potrebbero essere contenuti in una parola lunga 4 bit.



Con figurazione di 4 bit	Valore decimale
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Risulta quindi ovvio perché in genere si scelgono lunghezze di parola maggiori di 4: una macchina capace di rappresentare soltanto i numeri da 0 a 15 è ben lungi dall'essere adeguata. Ci sono però altri due problemi: questa notazione non è in grado di rappresentare valori non interi (per esempio 7.14) e non è in grado di rappresentare i numeri negativi.
Trascureremo volutamente il problema dei numeri "decimali", perché la stragrande maggioranza dei programmi in linguaggio macchina usa solo numeri interi, ma il problema di come rappresentare i numeri negativi è più urgente.
Il metodo impiegato è relativamente semplice: se si dispone della rappresentazione binaria di un numero positivo e si vuole determinare il suo equivalente negativo, basta fare due cose:
1. cambiare tutti gli 0 in 1, e tutti gli 1 in 0 (è quello che comunemente si dice "invertire tutti i bit");
2. aggiungere 1 al risultato (con le regole della somma binaria).
Per esempio, supponiamo di voler ricavare il binario per –3.

3 = 0011 come parola di 4 bit.

L'inversione di tutti i bit dà:	1100
Sommiamo 1:	+ 1
	1101

Dunque, 1101 rappresenta –3. Viene anche chiamato *complemento a 2* di 0011.
Non sto a spiegarvi esattamente come la cosa funziona, ma potete provare da soli che è così in tutti i casi, anche particolari, sulla base dell'esempio che segue.
Se sommiamo 3 a –3 (o anche 5 a –5, o un valore qualsiasi al suo opposto) si deve ottenere zero. E allora:

	0011	(= 3)
+	1101	(=–3)
=	10000	(Non dimenticare che 1 + 1 = 0 con riporto di 1 in binario!)

Come si vede, *non* è che si ottenga proprio 0000: ma i 4 bit "inferiori" o "più bassi" (come anche si dice) *sono* degli zeri, e dato che stiamo operando con parole di 4 bit, il bit di posto maggiore casca.
(Per un'analogia che vi sarà abbastanza nota, considerate un contachilometri sul cruscotto di un'auto: quando segna 99999 e viene percorso un ulteriore chilometro, la lettura passa a 00000: l'1 di testa "sparisce".)
In base a questa analogia, l'esempio precedente si può vedere anche così:

```
     [0011]
 +   [1100]
 ----------
 ┌───[0000]
 ↓
 1
```

Il tutto funziona in ogni caso, purché il numero di cifre sia fisso. Non scordate di premettere gli zeri non significativi per portare il numero totale di bit alla lunghezza prevista per una parola, *prima* di effettuare il complemento a 2.
E ora proviamo a riscrivere la tabella dei valori a 4 bit, stavolta includendo i numeri negativi:

Decimale	Binario	Complemento a 2	Decimale
0	0000	0000	0
1	0001	1111	−1
2	0010	1110	−2
3	0011	1101	−3
4	0100	1100	−4
5	0101	1011	−5
6	0110	1010	−6
7	0111	1001	−7
8	1000	1000	−8
9	1001	0111	−9
10	1010	0110	−10
11	1011	0101	−11
12	1100	0100	−12
13	1101	0011	−13
14	1110	0010	−14
15	1111	0001	−15

Vediamo subito che sorge un problema: ogni schema di bit compare due volte, per cui, per esempio, 1001 può significare 9 oppure −7. Dobbiamo ulteriormente restringere il campo dei valori. Ho tracciato una linea tratteggiata attorno alla regione che in sostanza intendiamo rappresentare. Se date un'occhiata al bit più significativo (quello più a sinistra) di ogni configurazione potrete notare che se questo è 0 il numero è positivo, mentre se vale 1 il numero è negativo. Abbiamo così trovato una maniera comoda per distinguere le due categorie di numeri interi. Pertanto, il campo di numeri che possiamo rappresentare con una parola lunga 4 bit va da −8 a +7 (incluso lo 0). Se la lunghezza fosse di 5 bit si potrebbero rappresentare i numeri da −16 a +15; con 6 bit, i numeri da −32 a +31; e così via.

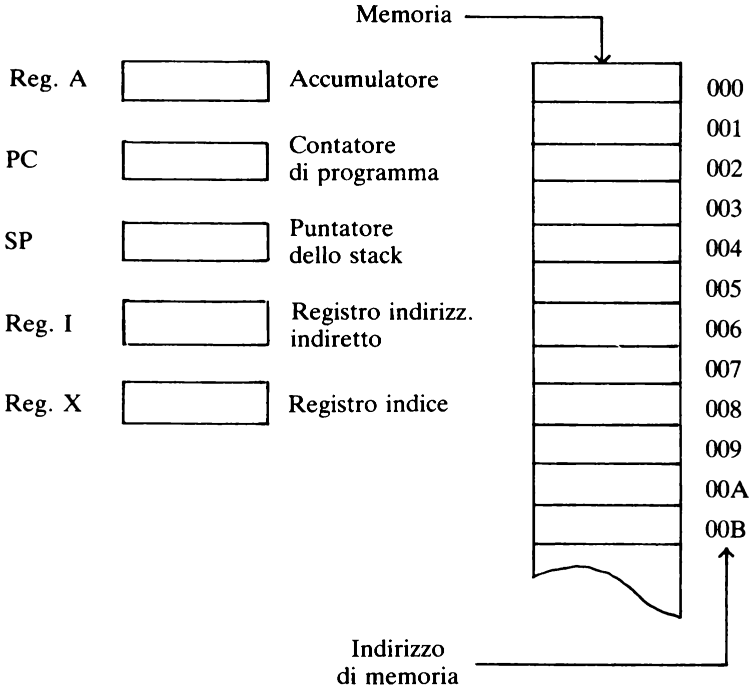
Con una parola lunga 16 bit (che è la lunghezza standard per lo Z 80) possiamo rappresentare numeri fra −32768 e +32767. Nell'Appendice 1 potete trovare una tabella dei complementi a 2 per parole di 8 bit.

CAPITOLO QUATTRO

Architettura della macchina

E per ora basta con i numeri. Ora daremo uno sguardo a come la macchina li "digerisce". Per fare questo, dobbiamo imparare qualcosa sulla struttura interna del microprocessore – ossia sulla sua *architettura*. Lo Z 80 è il risultato di circa 25 anni di sviluppo nel campo dei computer e relativi microprocessori, ed è un "animale" alquanto raffinato. Perciò non è proprio l'ideale per chi deve assimilare i primi concetti. Quello che intendo fare, quindi, è descrivervi un processore molto semplice, del tipo che avrebbe potuto venire costruito verso la fine degli anni Quaranta (in realtà non è stato proprio così, il progresso è stato più rapido), soltanto allo scopo di presentare i concetti più importanti, e validi anche per tutti i componenti attuali, senza dover badare troppo alle caratteristiche speciali, alle quali potremo dedicarci più tardi (lo faremo a partire dal capitolo 7).

Faremo l'ipotesi che la nostra macchina immaginaria abbia una memoria adatta a parole di 16 bit, e possegga un certo numero di registri speciali, come quelli descritti nella pagina a fronte.



Consideriamo per prima la memoria. In Basic avremmo potuto denominare ciascuna di queste locazioni di memoria nel modo che più ci fosse piaciuto, ma la nostra macchina ridotta all'osso non è così "amichevole". Essa insiste nel voler numerare ciascuna locazione in maniera assolutamente fissa, partendo da zero, come è mostrato in figura. Questi numeri vengono detti *indirizzi di memoria*, e io ho assegnato ad essi una numerazione in esadecimale, anche se non dovete dimenticare che, alla fin fine, la codifica risulterà binaria.

Che cosa può essere contenuto in una parola di questa memoria? Beh, una qualsiasi configurazione di 16 bit. Ovvio: ma quello che voglio fare notare è che questi 16 bit possono avere qualsiasi significato vogliamo attribuire loro. Se intendiamo che significhino un numero intero in complemento a 2, una parola può contenere allora un numero compreso fra –32768 e +32767. Se vogliamo attribuire loro il significato di numeri positivi senza segno, allora possono contenere un numero fra 0 e 65535. Se lo desideriamo, possiamo anche suddividere una parola in due campi di 8 bit ciascuno, ognuno dei quali serva a rappresentare un simbolo alfabetico, di punteggiatura o grafico. Come diceva Tweedledee (o era Tweedledum? Vedi a pag.): "Quando *IO* uso una parola, essa significa esattamente quello che ho scelto volesse significare – niente di più e niente di meno". Ho spesso pensato che Lewis Carroll precorresse i tempi.

Torniamo ai registri speciali. Cominciamo dal registro A per togliercelo subito di torno. Esso viene usato ogni volta che si eseguono calcoli aritmetici. Il risultato di una somma che per esempio fate calcolare alla macchina viene posto entro il registro A (che per questo riceve solitamente il nome di *accumulatore*, per inciso). Molte operazioni aritmetiche lavorano su due valori: non basta chiedere alla macchina di eseguire 3+, dovete indicarle a che cosa bisogna sommare il 3. Uno di questi valori deve essere presente nel registro A prima di eseguire l'operazione di somma. Pertanto si può scrivere un'operazione come questa:

ADD(1A3)

che la macchina intende così:

1. Somma il contenuto della locazione di memoria 1A3 al contenuto del registro A. (Le parentesi poste attorno a 1A3 sono un modo convenzionale molto usato per indicare che è il contenuto di 1A3 e non il numero 1A3 che va sommato.)
2. Poni il risultato ancora nel registro A.

Abbiamo così scritto la nostra prima istruzione in linguaggio macchina! Beh, non proprio in linguaggio macchina corretto, ma abbastanza vicino. Osservatene la forma generale. Essa consiste di un codice operativo, qui ADD, e di un indirizzo, qui (1A3). Molte istruzioni reali saranno simili a questa. Incidentalmente, la vita è troppo breve per insistere coll'usare il lungo termine "codice operativo": spesso lo si trova abbreviato in "opcode" (all'inglese).

UN PROGRAMMA PER LA SOMMA

Consideriamo una sequenza di istruzioni macchina che realizzino l'istruzione Basic

LET R = B + C

Per prima cosa dovremo assegnare degli effettivi indirizzi a R, B e C. Supponiamo che questi siano 103, 104 e 105 rispettivamente.
Dobbiamo porre il contenuto della locazione 104 nel registro A. Inventiamo allora un'istruzione LD (per *Load Accumulator*: carica nell'accumulatore) per fare questo:

LD(104)

Poi dobbiamo sommare il contenuto di 105

ADD(105)

ed infine dobbiamo trovare il modo di memorizzare il contenuto di A nella locazione 103. Inventiamo un'altra istruzione (ST per *store*, cioè deposita):

ST(103)

Ed ecco che abbiamo un semplice programma a livello macchina che consiste nelle tre istruzioni:

LD(104)	(carica B nel registro A
ADD(105)	(sommaci C)
ST(103)	(poni il risultato in R)

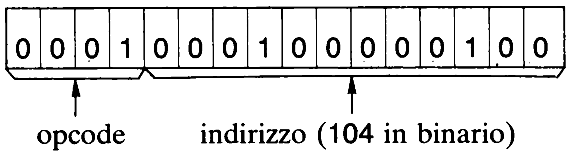
Come facciamo a dire alla macchina di eseguire questo programma? Noi siamo abituati all'idea che un programma è memorizzato nella macchina *prima* della sua esecuzione. Dopotutto, se scrivete un'istruzione Basic

10 PRINT "SALVE A TUTTI"

sareste alquanto sconcertati se, non appena premete ENTER, comparisse la scritta "SALVE A TUTTI" sullo schermo. Voi vi aspettate che il messaggio sia conservato entro la macchina sino al momento che lo richiedete. Così, allo stesso modo, occorre che un programma in linguaggio macchina venga per prima cosa memorizzato. Che c'è di più naturale come idea per memorizzare un'istruzione che depositarla in una parola di memoria? (Ricordate? "Una parola ha il significato che voi le attribuite".) Naturalmente, questo implica che gli opcode LD, ADD e così via vengano codificati come schemi di bit, ma tutto quello che ci serve a tal fine è di inventare una tabella di configurazioni di bit con i loro significati, in modo del tutto arbitrario come ad esempio questo:

Mnemonica dell'opcode	Codice binario
ADD	0000
LD	0001
ST	0010

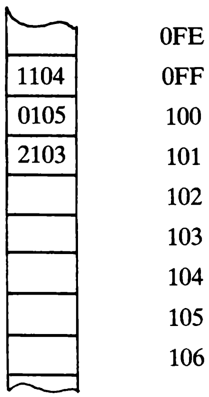
e ogni volta che inventeremo un nuovo opcode, non faremo altro che aggiungerlo nella tabella.
Qui ho supposto che tutti gli opcode abbiano un codice binario di 4 bit. La cosa consente 16 configurazioni diverse, e così 16 distinte istruzioni. Si tratta di un set di istruzioni veramente esiguo, riferito ai moderni standard in materia, ma per il nostro ipotetico computer può bastare. La parola è composta di 16 bit, cosicché ce ne rimangono liberi 12 per la parte indirizzo dell'istruzione.
Perciò LD(104), una volta inserito nella macchina, apparirà come



Una volta visto uno degli schemi di bit, visti tutti: perciò di qui in avanti scriveremo solo la versione esadecimale delle istruzioni. Fra l'altro, è anche un po' meno noioso.

IL CONTATORE DI PROGRAMMA

Supponiamo di aver memorizzato il nostro programmino di tre istruzioni a partire dall'indirizzo OFF in avanti:



Quello che ora ci serve è un modo per dire alla macchina: "Forza, parti eseguendo l'istruzione che si trova in OFF, poi quella in 100, poi quella in 101". A questo serve il registro PC, o contatore di programma (*Program Counter*). Esso funziona come una sorta di contatore di indirizzi per il computer. Il programma viene fatto partire inizializzando

il registro PC con l'indirizzo della prima istruzione. Mentre la macchina esegue questa istruzione, il PC viene automaticamente incrementato di 1, così quando il sistema torna a esaminare il contenuto del registro PC, passa a eseguire la successiva istruzione, di cui trova ivi l'indirizzo, e così via.

C'è un piccolo inconveniente, però. Quando viene eseguita l'ultima istruzione (posta in 101), il registro PC viene incrementato di 1 come al solito, e così quando la macchina passa a esaminarne il contenuto, vi trova 102, e passa ad eseguire l'istruzione ivi collocata. Quale istruzione? Alla locazione 102 non avevamo memorizzato alcuna istruzione! Però, qui si può comunque trovare una certa configurazione di bit lasciata lì da un programma precedente, o semplicemente creata al momento in cui la macchina è stata posta in funzione. E allora la macchina interpreterà questo schema di bit come se fosse un'istruzione, perché questo è quello che le abbiamo ordinato tramite il PC. E poi passerà analogamente alle locazioni 103, 104 e 105, ossia proprio quelle in cui abbiamo memorizzato (per ipotesi) i nostri dati! Se quindi ad esempio, il numero posto in 104 è 20FF, la macchina lo interpreterà come

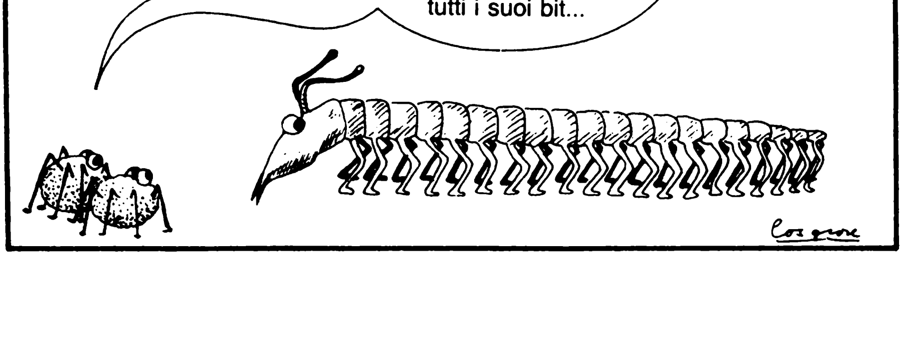
```
ST(0FF)
```

ricopiando il contenuto attuale del registro A entro la locazione 0FF, e così distruggendo la prima istruzione del nostro programma! Ovviamente, quello che qui ci serve è un'istruzione di STOP o ALT (la mnemonica usata di solito è HLT) che serva a fermare l'aggiornamento del registro PC a un certo punto del suo procedere. Ora il programma sarà dunque:

```
LD(104)
ADD(105)
ST(103)
HLT
```

Qui c'è un'importante osservazione da ricavare. Proprio *perché* usiamo parole che possono avere diverso significato in diversi momenti, dobbiamo sempre stare molto attenti alle implicazioni che la macchina può trarre da quello che le diciamo di fare. Se desideriamo che essa sommi (ADD) il contenuto di una certa locazione a quello del registro A, essa assumerà che quella locazione contenga un numero. Non verrà eseguito alcun controllo: e in pratica non si può fare – ogni configurazione di bit può benissimo rappresentare anche un numero. Similmente, ogni configurazione di bit può rappresentare un'istruzione, cosicché se il PC

punta a una locazione, il relativo contenuto verrà considerato un'istruzione e come tale eseguito, una volta interpretato.
Da qui la regola: *tenete ben separati dati e programmi*. Se non lo fate, può capitarvi di tutto, in certi casi. Come abbiamo appena visto, può succedere che un intero programma scompaia senza lasciar tracce mentre viene eseguito!



CAPITOLO CINQUE

Salti e sottoprogrammi

Per ora il gruppo delle nostre istruzioni è un po' scarno. Abbiamo LD e ST, che servono per muovere qualcosa entro la memoria, ADD, che è una forma assai primitiva di calcolo aritmetico, e possiamo arrestare il programma con HLT.
Ora incrementeremo un tantino le capacità aritmetiche aggiungendo SUB, che servirà a sottrarre dal registro A il valore contenuto in una data locazione, ma questo è tutto. Non abbiamo istruzioni per il prodotto, né per la divisione, né tanto meno per estrarre una radice quadrata. Quello che in pratica ci serve è un certo numero di istruzioni di salto, equivalenti alle IF ... THEN del Basic.

SALTI

Saltare a un'istruzione che non sia nella sequenza normale è cosa relativamente facile: ci basterà modificare il contenuto del registro PC (contatore di programma). Useremo quindi un'istruzione tipo

JP 416 (esegui un salto alla locazione 416)

Ogni volta che essa verrà eseguita, per prima cosa provvederà a collocare il valore 416 entro il PC. In questo modo il sistema verrà "imbrogliato" e penserà che la prossima istruzione si trova all'indirizzo 416, dopodiché passerà nell'ordine alle locazioni 417, 418 ecc., sino a che non venga incontrata un'altra istruzione di salto. Come è ovvio, all'opcode JP può far seguito un indirizzo qualunque.

In effetti, questa istruzione assomiglia più a un GO TO che a un IF...THEN. Ci serve per questo un'istruzione che modifichi il contenuto di PC solo se è soddisfatta una certa condizione. Il controllo più facile da effettuare consiste nel verificare se il registro A contiene lo 0:

JPZ2A7 (salta a 2A7 se il registro A è Zero).

Un'istruzione analoga potrebbe essere:

JPN14E (salta a 14E se il contenuto di A è Negativo)

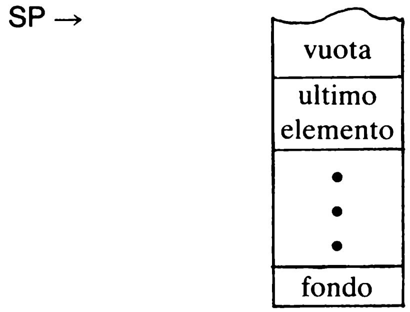
Questo è il minimo indispensabile, perché ora siamo in grado di verificare se un numero è positivo (non-zero) controllando se il programma *non* effettua nessuno dei due salti corrispondenti a JPZ e JPN.

I SOTTOPROGRAMMI E LO STACK

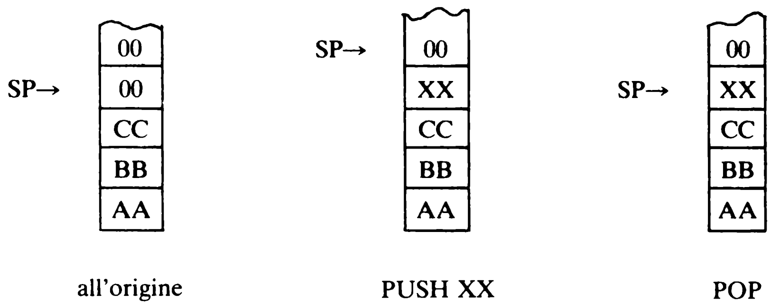
Dato che stiamo parlando dei modi per trasferire il controllo da un punto a un altro del programma, che cosa si può dire a proposito di equivalenti dei GOSUB e RETURN del Basic?
Creiamo un'istruzione

CALL205 (chiama il sottoprogramma che inizia da 205)

Che cosa fa questa istruzione? Bene, ovviamente comincerà col porre 205 entro il PC, ma allo stesso scopo avremmo potuto adoperare un JP. CALL in effetti svolge anche un'altra funzione: memorizza l'indirizzo dell'istruzione immediatamente successiva a quella del CALL, di conseguenza quando viene incontrato un RETURN (opcode: RET), può procedere a ricaricare nel PC l'indirizzo prima memorizzato, per riprendere l'esecuzione del programma principale dal punto in cui l'aveva lasciato. Il tutto si ottiene con l'ausilio di uno *stack* (termine inglese molto usato, e in genere migliore della traduzione italiana in "pila" anche se è difficile fare confusione con una batteria...). Lo stack è una porzione di memoria con un "fondo" fisso e una cima variabile. Un registro "puntatore dello stack", SP (*Stack Pointer*) contiene in ogni momento l'indirizzo della cima (variabile) dello stack: viene chiamato puntatore proprio perché "punta", come una freccia indicatrice, alla cima dello stack, a questo modo:



Nuovi elementi possono venire spinti (PUSH in inglese) sulla cima dello stack incrementando di uno lo SP, e collocando il nuovo elemento alla locazione posta in cima allo stack; e i vari elementi "impilati" possono venire estratti (POP in inglese) dalla cima dello stack decrementando lo SP di uno. (Non è strettamente necessario cancellare gli elementi estratti: le routine riguardanti lo stack semplicemente ignorano tutto quello che si trova al di sopra della locazione correntemente puntata dallo SP.) Per esempio:



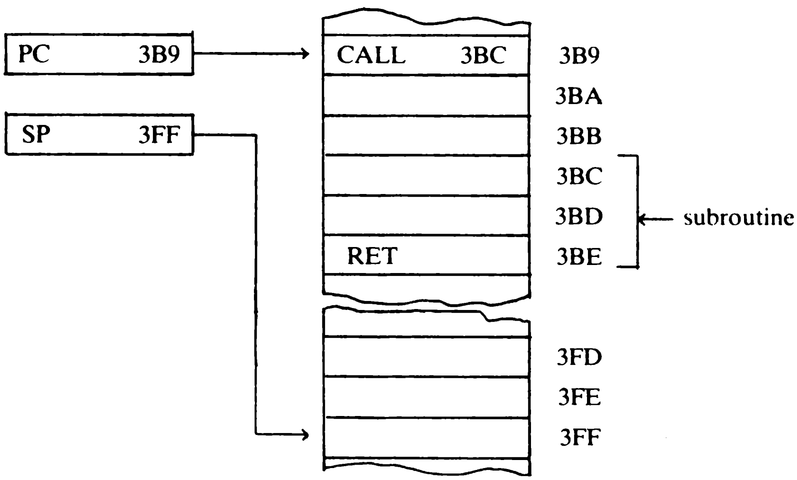
In effetti, lo SP punta in ogni momento alla prima locazione *non utilizzata* dell'area dello stack. (Inoltre, bisogna ricordare che lo stack macchina dello Z 80 "cresce" verso il basso, non verso gli indirizzi più alti: ma non preoccupatevene, perché per l'utilizzatore la cosa non fa alcuna differenza, tanto in pratica non importa affatto che conosca gli effettivi indirizzi nello stack, che sono gestiti internamente dalla macchina.)
Lo stack funziona in base al principio che un popolare acronimo inglese definisce LIFO, ossia "*Last In First Out*", "esce per primo chi è entrato per ultimo". Provate a immaginare una pila di libri uno sopra l'altro su un tavolo. PUSH può significare "poni un altro libro sopra gli altri", e POP, analogamente "togli dalla cima della pila il libro più in alto". Così se si effettua successivamente il PUSH di tre elementi X, Y e Z

PUSH X
PUSH Y
PUSH Z

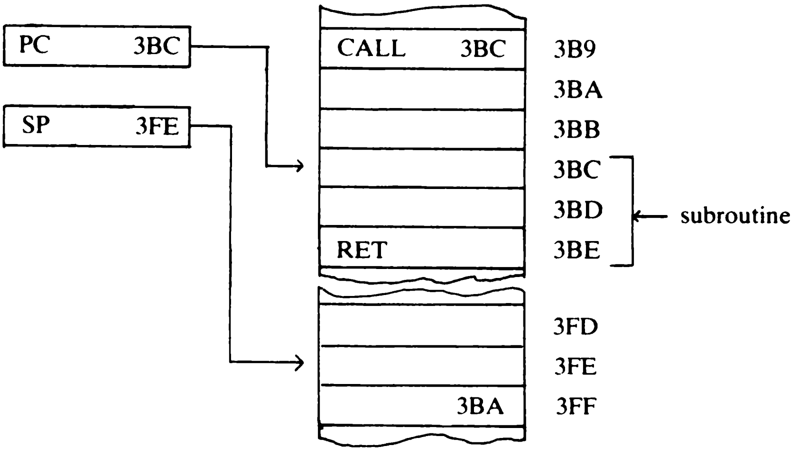
per riottenerli nel giusto ordine bisogna eseguire

POP Z
POP Y
POP X

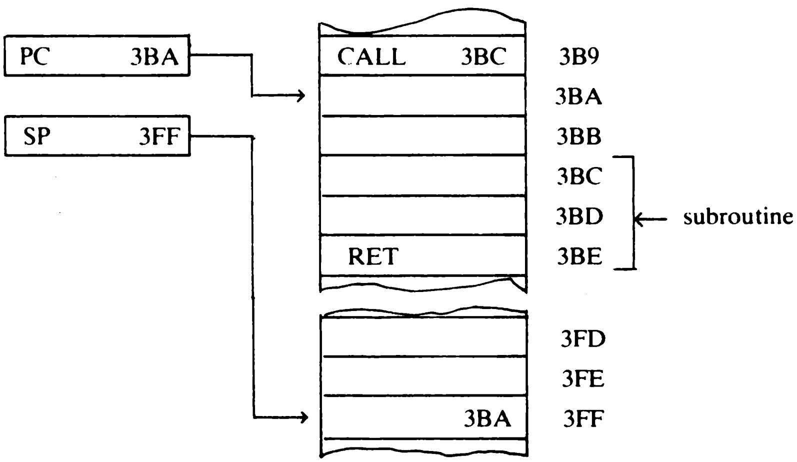
Un sottoprogramma (o subroutine, come è anche spesso chiamato) utilizza lo stack per ricordare quale è l'indirizzo di ritorno. Quando si obbedisce a un'istruzione CALL, l'indirizzo di ritorno (ossia l'indirizzo dell'istruzione CALL + 1) viene collocato in cima allo stack. Quando si incontra una RET, il valore in cima allo stack viene estratto (POP) e inserito nel PC. Ecco un esempio:



Si sta per eseguire l'istruzione CALL...



E ora è stata eseguita e l'indirizzo di ritorno si trova in cima allo stack. Il programma procede attraverso la subroutine sino a raggiungere la **RET**, dopo di che:



e il controllo viene restituito al programma principale.

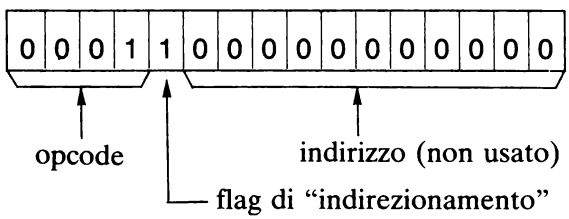
CAPITOLO SEI

In-direzionamento e indicizzazione

Ci sono solo due registri di cui non abbiamo ancora parlato, che svolgono entrambi funzioni simili: entrambi sono capaci di modificare la parte indirizzo di un'istruzione mentre si sta eseguendo il programma.

IN-DIREZIONAMENTO (INDIRIZZAMENTO INDIRETTO)

Vediamo come questo viene fatto cominciando dal registro I. Inventiamo un nuovo opcode, LDI ovvero "carica in modo indiretto". Come HLT, questa istruzione non ha un indirizzo associato. Per la macchina è la stessa cosa che LD, salvo che il bit più "alto" (significativo, ovvero posto più a sinistra) del campo indirizzo è posto a 1. Tale bit viene chiamato *flag dell'indirizzamento indiretto*, e serve unicamente a ricordare alla macchina che è in atto questo modo di indirizzamento (potremmo chiamarlo "in-direzionamento"). La forma binaria dell'istruzione LDI è dunque



Il codice esadecimale relativo è 1800. Quando la macchina incontra questa istruzione, utilizza il contenuto che comunque si trova nel registro I come indirizzo effettivo. Se il registro I contiene 1E4 e viene eseguita un'istruzione LDI, l'effetto è esattamente lo stesso di un'istruzione LD1E4. In altre parole, il registro I funge da puntatore entro la memoria, e possiamo fargli puntare dove più ci piace, se questo può aiutarci nei calcoli. Il che vuol dire far caricare entro il registro A valori provenienti da varie locazioni, dato che questo è l'unico registro adibito ai calcoli aritmetici.
E perciò inventiamo anche un opcode XAI per "scambia fra di loro i contenuti dei registri A e I" (X sta per *exchange*).
Ovviamente, il flag dell'indirizzamento indiretto può venire posto a 1 anche in ogni altra istruzione che abbia una parte indirizzo. Potremmo avere così istruzioni STI, JPI, ADDI ecc., e in ogni caso le tre ultime cifre dell'opcode saranno 800.

UN ESEMPIO

Vediamo un esempio che sfrutta questa idea. Supponiamo di voler inizializzare un vettore (variabile multipla unidimensionale) composto di 20 elementi, con i valori 2, 4, 6, 8, ..., 40. In altre parole, vogliamo un'equivalente in linguaggio macchina del programmino Basic

```
FOR c = 1 TO 20
LET a(c) = c*2
NEXT c
```

Per fare questo occorre che nella memoria siano presenti un certo numero di valori, in locazioni idonee. Si tratta dei valori 1 (perché il ciclo procede per passi di 1), 2 (perché questo è l'incremento fra i successivi elementi del vettore) e 20 (che serve a verificare se si è concluso il ciclo). Per ora trascuriamo dove esattamente verranno memorizzati questi numeri: per indicarli in modo temporaneo userò dei nomi (proprio come nel Basic, con i nomi delle variabili). Quando passeremo al linguaggio macchina definitivo dovremo naturalmente convertire questi nomi convenzionali nei numeri relativi. Si tratta di un'altra applicazione della famosa Prima Legge del Calcolo di Jones: "Mai rimandare a domani quello che si può rimandare a dopodomani". Supporremo dunque che i numeri che ci servono siano reperibili in locazioni dette N1, N2 e N20. Avremo inoltre una locazione detta BASE che conterrà l'indirizzo del primo elemento del vettore, e una detta CONTO che servirà da contatore dei cicli.

Per prima cosa faremo in modo che il registro I punti all'indirizzo di base del vettore:

```
LD BASE
XAI
```

Poi posizioniamo CONTO a 1:

```
LD N1
ST CONTO
```

Ora, raddoppiamo questo valore (per semplice somma a se stesso entro il registro A), e lo memorizziamo nella locazione a cui punta il registro I. (In altri casi simili adopereremo la locuzione "memorizzare tramite il registro I" per brevità.)

```
ADD CONTO
STI
```

Ora "eliminiamo" il raddoppio nel registro A, sottraiamo 20 per verificare se il risultato vale zero. Se è così, abbiamo terminato:

```
SUB CONTO
SUB N20
JPZ OUT
```

dove OUT è un altro indirizzo non specificato. Per ora non lo conosciamo, perché non sappiamo dove terminerà il programma, e per questo torna ancora comodo assegnargli un nome temporaneo.
Se il salto non viene eseguito, incrementiamo CONTO di 1:

```
LD CONTO
ADD N1
ST CONTO
```

e incrementiamo il registro I pure di 1:

```
XAI
ADD N1
XAI
```

Il valore corrente di CONTO si trova ora nuovamente nel registro A, cosicché possiamo tornare indietro all'operazione di raddoppio:

```
JP LOOP
```

ammesso che all'istruzione ADD CONTO abbiamo assegnato l'indirizzo simbolico LOOP (ciclo). Realizziamo praticamente la cosa premettendo all'istruzione il suo indirizzo simbolico seguito da due punti (:)

LOOP: ADD CONTO

Possiamo fare analogamente per posizionare i valori iniziali che ci servono, definendo un nuovo opcode HEX, che serve ad assegnare a una data "parola" un valore determinato. Non si tratta in realtà di un vero opcode, perché non equivale a un'istruzione macchina: lo chiameremo pseudo-opcode. L'intero programma assumerà l'aspetto seguente (per ora trascurate i numeri della prima e ultima colonna):

```
020           LD    BASE    1 033
021           XAI           A 000
022           LD    N1      1 030
023           ST    CONTO   2 032
024 LOOP:     ADD   CONTO   0 032
025           STI           2 800
026           SUB   CONTO   4 032
027           SUB   N20     4 031
028           JPZ   OUT     6 047
029           LD    CONTO   1 032
02A           ADD   N1      0 030
02B           ST    CONTO   2 032
02C           XAI           A 000
02D           ADD   N1      0 030
02E           XAI           A 000
02F           JP    LOOP    5 024
030 N1:       HEX   0001    0 001
031 N20       HEX   0014    0 014
032 CONTO:    HEX   0000    0 000
033 BASE      HEX   0000    0 000
```

L'unico indirizzo simbolico che non appare nella colonna a sinistra, e quindi ancora non specificato, è OUT. Ce ne occuperemo dopo.
Il programma nella forma in cui l'abbiamo appena scritta viene chiamato *codice in assembly* o *codice sorgente*. Nei computer moderni e sofisticati sarà quasi sempre disponibile un programma *Assembler* con il

compito di convertire un programma di questo tipo nel vero e proprio linguaggio macchina, per conto nostro.

"ASSEMBLAGGIO" A MANO

Peccato che la nostra ipotetica macchina, così come lo Spectrum di dotazione, non disponga di un simile programma (anche se in commercio ne esistono diversi sotto forma di cassette). Allora dobbiamo procedere a fare il lavoro a mano. Abbiamo bisogno di una tavola di opcode con a fronte i relativi equivalenti esadecimali:

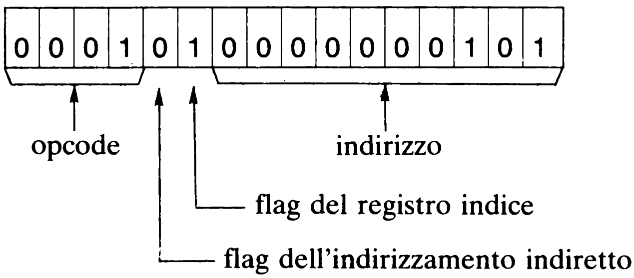
Opcode	Esadecimale
ADD	0
LD	1
ST	2
HLT	3
SUB	4
JP	5
JPZ	6
JPN	7
CALL	8
RET	9
XAI	A

Dobbiamo anche conoscere dove sta l'inizio del programma. Qui la decisione può essere alquanto arbitraria, pertanto supponiamo di averlo posto alla locazione 020. Dato che (nel nostro ipotetico sistema) ogni istruzione occupa una parola, possiamo scrivere gli indirizzi corrispondenti a ciascuna istruzione. Sono quelli riportati nella prima colonna del programma appena listato. Ora possiamo sostituire opcode e indirizzi con i corrispondenti equivalenti esadecimale.
Per esempio, LD BASE diventa 1 033, perché ora abbiamo identificato BASE come 033. Nell'ultima colonna del listato abbiamo riportato appunto il codice esadecimale completo.
L'unica istruzione che richiede un chiarimento è JP OUT, codificata come 6 047. Perché OUT avrebbe il valore 047? Difatti, potrebbe essere un altro, ma qui 047 è la prima locazione dove *potrebbe* cadere. Il motivo sta nel fatto che il vettore occupa lo spazio da 033 a 046 (venti

parole, venti elementi), e ovviamente non fisseremo un valore di indirizzo che possa interferire entro l'area dei dati del nostro programma.

IL REGISTRO INDICE

Quando viene usato il registro X, l'indirizzo effettivo dell'istruzione viene formato aggiungendo al valore contenuto nel campo indirizzo di una certa istruzione il valore contenuto nel registro X.
Per esempio, se il registro X contiene 400, l'istruzione LDX005 ha lo stesso effetto della LD 405.
Distingueremo un altro bit entro il campo indirizzo quando è in atto l'"indicizzazione", e così l'istruzione citata, LDX 005, assumerà questo aspetto:



In esadecimale, lo leggiamo come 1405. In pratica, tutto quello che si può fare coll'indirizzamento "indicizzato", si può fare altrettanto bene con l'indirizzamento indiretto. Comunque, i calcoli aritmetici con gli indirizzi vengono da questi svolti automaticamente, invece di far fare il lavoro a voi.



CAPITOLO SETTE

Lo Z 80 – finalmente!

Mi spiace di avervi propinato le ultime dieci (o pressappoco) pagine senza permettervi di sperimentare alcunché, comunque se avete effettivamente compreso i concetti in esse esposti, vedrete che capire lo Z 80 diventa uno scherzo.

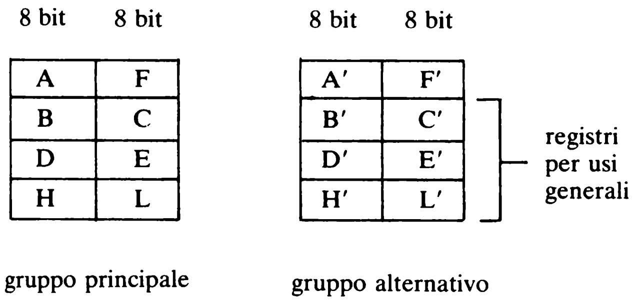
Prima di addentrarci nell'architettura dello Z 80 (spiacente, il titolo forse non è ancora esatto!), vediamo di esaminare alcune difficoltà che riguardano l'ipotetico processore che ho appena finito di descrivervi. Per prima cosa, un codice operativo su 4 bit ci consente solo un massimo di 16 istruzioni (OK, abbiamo barato leggermente consentendo che i flag dell'indirizzamento indiretto e indicizzato si infilassero entro il campo riservato agli indirizzi, ma ciò vuol dire a sua volta che abbiamo limitato le dimensioni dell'indirizzo, e quindi le dimensioni massime della memoria!). Ora, lo Z 80 possiede 694 istruzioni! Per assegnare a ciascuna di esse una diversa configurazione di bit significa che ci serve un campo di 8 bit (un byte); ed anche così bisogna arrangiarsi un poco.

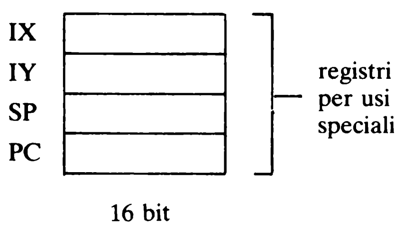
Secondo, la nostra macchina immaginaria utilizza la memoria in una maniera alquanto "trascurata". Alcune istruzioni non impiegano il campo indirizzi (come HLT, LDI, STI, per esempio), e una sequenza di tali istruzioni spreca 10 bit per ogni parola. Lo Z 80 supera questo problema ammettendo che le diverse istruzioni abbiano lunghezza differente. Alcune istruzioni non hanno campo indirizzi, e quindi occupano un singolo byte. Altre hanno un campo indirizzi di un solo byte, e sono quindi lunghe 2 byte. Altre ancora hanno un indirizzo lungo due byte, per un totale di 3 byte. Ce ne sono addirittura alcune con opcode di 2 byte! Questo significa che non si può incrementare il PC di 1 per

ogni tipo di istruzione eseguita: bisogna che l'incremento sia pari alla lunghezza di tale istruzione.
Terzo, dobbiamo sempre manipolare parole lunghe 16 bit, il che risulta scomodo se abbiamo a che fare con i caratteri (che solitamente occupano un singolo byte). Pertanto sarebbe giusto poter ammettere operazioni *sia* a 8 *sia* a 16 bit.
Quarto, e ultimo, il fatto che si disponga solo di un unico registro tuttofare (il registro A) può risultare fastidioso. Ciò vuol dire che spesso i risultati intermedi devono venire temporaneamente memorizzati a parte, mentre vengono eseguiti altri calcoli. Lo Z 80 dispone di un certo numero di registri per usi generali; anche se, come vedremo, il numero esatto di quelli disponibili varia a seconda dell'uso a cui sono destinati.

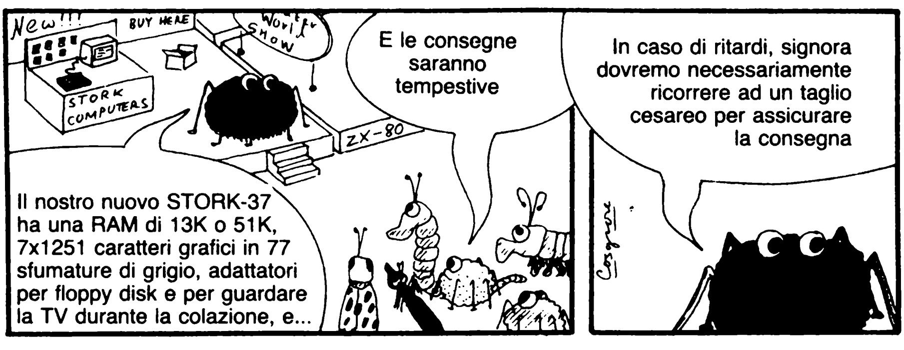
I REGISTRI

Eccovi l'organizzazione dei registri dello Z 80:





Per il momento, dimenticate il set di registri alternativi.
I registri appaiono in coppie, indicando che essi possono venire impiegati sia come registri di 8 che di 16 bit. Per esempio, possiamo riferirci al registro B (di 8 bit), al registro C (pure di 8 bit), oppure al registro BC (di 16 bit). I registri B, C, D, E, H e L possono venire impiegati tutti in questo modo (le coppie che danno registri a 16 bit sono *solo* le BC, DE e HL); mentre i registri A e F sono strettamente di 8 bit, e non possono venire combinati assieme. Nei registri usati in coppia, il byte "alto" o più significativo (MSB = *Most Significant Byte*) è in quello di sinistra (B, D o H), mentre il byte "basso" (LSB = *Least Significant Byte*) è nell'altro, come probabilmente vi attendevate.
Ci sono pure due registri indice, IX ed IY, uno stack pointer SP e il contatore di programma PC. Ma come, nessun registro per l'indirizzamento indiretto? In effetti, ciascuno dei registri-coppia a 16 bit (BC, DE, HL) può venire impiegato per l'in-direzionamento, ma, per semplicità, finiremo per usare sempre il registro HL per questo scopo.
Vi sono due gruppi di istruzioni, uno relativo alle operazioni per il trattamento di 8 bit, e l'altro per 16 bit. Inizieremo con le istruzioni di caricamento (LD, *load*) a 8 bit.



CAPITOLO OTTO

Modi di indirizzamento e le istruzioni LD

Diamo un'occhiata all'operazione di "caricamento" (LD), come esempio del gruppo di istruzioni a 8 bit. Assomiglia da vicino all'istruzione LD della nostra macchina immaginaria, salvo che sono ammessi altri due modi di indirizzamento: *da registro a registro* e *immediato*. Con i modi *diretto, indiretto* e *indicizzato* già citati, fanno un totale di 5.

1. *Indirizzamento diretto*

È molto simile a quello concepito per la nostra macchina ideale, eccetto per il fatto che, dato che esiste più di un registro, occorre in ogni caso specificare in quale registro si vuole caricare:

LD A, (0F1C)

significa che nel registro A verrà caricato il contenuto della locazione 0F1C. Notare come, per convenzione, si intende sempre che il movimento vada da destra verso sinistra, per cui se scriviamo

LD(0F1C), A

si intende "carica il contenuto di A nella locazione 0F1C".
(In effetti, il registro A è l'*unico* registro a 8 bit che può venire indirizzato direttamente.)

2. *Indirizzamento indiretto*

Anche qui tutto è abbastanza chiaro. Dato che abbiamo intenzione di "standardizzarci" sul registro HL per l'indirizzamento indiretto, il formato dell'istruzione è

LD A, (HL)

che significa "carica il registro A *tramite* (vale a dire, dall'indirizzo contenuto in) il registro HL". Per passare dati nel senso opposto si scriverà

LD(HL), A

che serve a porre il contenuto di A nella locazione il cui indirizzo è *contenuto* in HL. (Per questa istruzione, sono ammessi registri diversi da A.)

3. *Indirizzamento indicizzato*

Dobbiamo in questo caso segnalare quale registro indice deve essere impiegato, nonché il valore dello "scostamento" (*offset*):

LD A, (IX+2E)

Osservate come nel caso dell'indirizzamento diretto abbia usato un indirizzo di 4 cifre esadecimali, perché per gli indirizzi sono ammessi (anzi sono comuni) 16 bit (2 byte). Il valore dell'offset in un'istruzione di indirizzamento indicizzato invece è contenuto in un unico byte, ragion per cui ho usato solo due cifre esadecimali.

4. *Da registro a registro*

Possiamo trasferire dati da un registro a un altro, così:

LD D, B

che significa "carica nel registro D il contenuto del registro B". (Useremo spesso in seguito la dizione abbreviata "carica in D il contenuto di B", quando non addirittura "carica B in D".)

5. *Caricamento immediato*

Qui troviamo direttamente un dato, invece di un indirizzo, nella posizione solitamente riservata nell'istruzione all'indirizzo. Possiamo così scrivere

LD B, 07

a significare "carica il valore 7 in B". Notate anche qui che il valore è un numero esadecimale di due cifre, dato che deve essere collocato nel singolo byte del registro B. Notate inoltre come in ogni caso l'istruzione LD significhi in realtà "copia": infatti i numeri posti agli indirizzi o nei registri originali sono conservati, mentre è una loro copia che viene posta a destinazione.

CODICI ESADECIMALI

Diamo ora uno sguardo a come queste istruzioni vengono espresse in esadecimale; per un elenco completo vi rinviamo all'Appendice 6.

1. LD A,(0F1C)

Prima cerchiamo l'opcode per l'istruzione tipo LD A,(nn) (dove nn indica un generico indirizzo di 2 byte). Troviamo 3A. Per cui si può ritenere che la codifica dell'istruzione sia

3A 0F1C

Purtroppo c'è una piccola complicazione, che deriva dalla maniera con cui lo Z 80 tratta i numeri: infatti esso preferisce avere il byte meno significativo ("basso") dell'indirizzo per primo. Di conseguenza occorre *scambiare fra loro i due byte dell'indirizzo*:

3A 1C0F

La cosa è relativamente fastidiosa, ma ci si fa presto l'abitudine.
Si tratta di una norma fissa per i numeri di 2 byte nelle istruzioni dello Z 80: *prima il byte basso, poi quello alto*; questo è il motivo dei diversi PEEK X + 256*PEEK(X+1) che troviamo nel Manuale d'uso Sinclair.
L'istruzione "inversa" (LD(nn), A) ha per opcode 32, pertanto

LD(0F1C), A   diventa in esadecimale 32 1C0F.

2. LD A,(HL)

Facile. Qui non abbiamo alcun indirizzo, così tutto si limita al singolo opcode di 1 byte. Cercatelo, e troverete il codice 7E.
Analogamente, l'istruzione LD(HL), A ha come opcode il singolo 77.

3. LD A,(IX+2E)

La generica istruzione ha la forma LD A,(IX+d), dove d indica il valore (1 byte) dello "spostamento" o offset, che (si badi) è in notazione con complemento a 2. Il suo codice è DD 7E (notate – si tratta di un opcode a due byte!). L'istruzione completa si traduce allora così:

DD 7E 2E

dove il particolare valore 2E è il valore dell'offset scelto in questo caso.

4. LD D, B

Anche qui, nessun problema: opcode a singolo byte; codice 50.

5. LD B, 07

L'opcode è 06, e facilmente ne segue che l'istruzione completa si scrive 06 07.

CAPITOLO NOVE

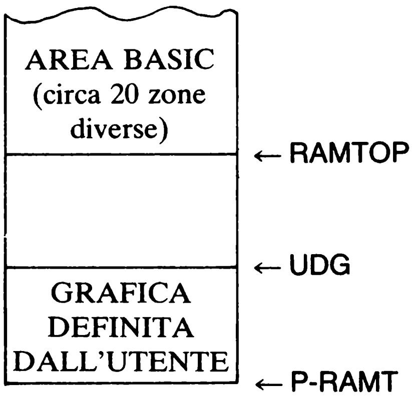
Memorizzazione, esecuzione e salvataggio del linguaggio macchina

Lo scopo primo di questo capitolo è la messa a punto di un semplice programma di "utilità" in Basic che ci sollevi da alcune delle pene legate al linguaggio macchina. Troverete una versione più sofisticata dello stesso programma nell'appendice 7.
Cominciamo col considerare la memoria dello Spectrum. È di due tipi:

Tipo	Indirizzo iniziale		Indirizzo finale	
	Decimale	Esadecimale	Decimale	Esadecimale
ROM	0	0	16383	3FFF
RAM	16384	4000	32767 65535	7FFF (16 K) FFFF (48 K)

La ROM contiene il sistema operativo, l'interprete Basic, e altro ancora: sarebbe disastroso se si potesse metterci le mani sopra, così è stata costruita in modo che questo non possa assolutamente succedere: ROM significa infatti *Read Only Memory*, ossia "memoria di sola lettura". In essa si possono fare dei PEEK, ma non dei POKE.
La RAM (*Random Access Memory*, memoria ad accesso casuale) è invece riservata a noi, per farci quello che più ci piace: ed è nella RAM che va memorizzato, e dove va eseguito, il linguaggio macchina. Un problema potenziale sta nel fatto che anche il Basic utilizza la RAM: se piazziamo il

nostro linguaggio macchina in una posizione "comune" della RAM, è sempre possibile che il Basic ce lo cancelli, o lo sovrascriva, o in genere possa modificarlo con gravi interferenze. Abbiamo dunque bisogno di salvare il nostro linguaggio macchina in un posticino che non venga usato (o almeno, che non possa venire alterato) dal sistema Basic.
Il posto più "simpatico" per fare questo (come vedremo, ce ne possono essere altri) è la parte alta della memoria. Eccovi un quadro della parte di memoria interessata.



Gli indirizzi dei limiti fra le varie regioni sono contenuti in tre variabili di sistema, i cui indirizzi sono eguali tanto per le macchine che dispongono di 16 K che di 48 K di RAM:

RAMTOP 23730-1
UDG 23675-6
P-RAMT 23732-3

Si tratta di variabili a due byte, il cui valore decimale si può determinare, per esempio, per il caso di RAMTOP, così:

```
PRINT PEEK 23730 + 256*PEEK 23731
```

Quando la macchina viene accesa per la prima volta, le tre variabili vengono inizializzate ai loro valori "normali":

Variabile	16 K		48 K	
	Decimale	Esadecimale	Decimale	Esadecimale
RAMTOP	32599	7F57	65367	FF57
UDG	32600	7F58	65368	FF58
P-RAMT	32767	7FFF	65535	FFFF

Il sistema Basic utilizza l'area posta al disotto e *inclusa* la locazione il cui indirizzo è in RAMTOP. La parte compresa fra UDG e P-RAMT (fine "fisica" della RAM), inclusa, serve a contenere i formati degli *user defined graphics* (caratteri grafici definibili dall'utente). Questa particolare area può venire ridotta, o del tutto eliminata modificando il valore di UDG; tuttavia, per non creare ulteriori confusioni, supporremo di conservare gli *user defined graphics* nell'area loro riservata.
Possiamo invece fare posto per del linguaggio macchina abbassando il valore di RAMTOP. Per esempio, se vogliamo liberare 600 byte di spazio nella zona alta della memoria, facendo partire il linguaggio macchina dall'indirizzo 32000, dobbiamo modificare il valore di RAMTOP in 32000 – 1 = 31999.
(In genere, questo è uno spazio più che sufficiente per tutti i programmi descritti in questo libro, col vantaggio di avere una cifra tonda per l'indirizzo di partenza sia in decimale (32000) che in esadecimale (7D00), per cui ho scelto di adottare di norma questo valore.
Se desiderate comunque un valore diverso, basterà che ovunque troverete riferimenti a 32000 e 31999, ovvero 7D00 e 7CFF, sostituiate il valore che avrete scelto e il suo precedente. Per non creare ulteriori diversità, ho scelto inoltre di adottare il valore 32000 *anche* per la macchina a 48 K; però è evidente che in questo modo ben 32 K di memoria vanno sprecati. Voi potreste perciò preferire di usare per questa macchina il valore 64000, che vale FA00 in esadecimale.
Mentre si sta *imparando* il linguaggio macchina, gli sprechi di memoria non hanno importanza: è evidente però che in programmi reali voi tenderete a scelte di RAMTOP che risultino più efficienti.)
Per abbassare RAMTOP, sarà sufficiente il seguente comando diretto:

```
CLEAR 31999
```

che comporta tutta una serie di effetti, e precisamente:
- cancella i valori di tutte le variabili
- cancella [la memoria del]lo schermo, allo stesso modo di CLS
- riposiziona il pixel in modo grafico ad alta risoluzione all'estremo inferiore sinistro (coordinate: 0,0)
- esegue un RESTORE che riporta all'inizio del primo DATA esistente
- cancella lo stack dei GOSUB, che conteneva gli indirizzi di ritorno delle subroutine sino allora utilizzate
- riposiziona (reinizializza) RAMTOP al valore 31999, rendendo le locazioni da 32000 in su "sicure" da ogni interferenza Basic
- posiziona il nuovo stack dei GOSUB immediatamente sotto il nuovo valore di RAMTOP.

Un analogo comando tipo

CLEAR 31447

farebbe esattamente lo stesso, reinizializzando però RAMTOP al valore 31447, lasciando libero uno spazio alquanto maggiore: tutte le locazioni da 31448 in su sarebbero salvaguardate (anche da un eventuale NEW). Naturalmente, il momento migliore per usare CLEAR a questo modo è prima che sia stato assegnato il valore ad una qualsiasi variabile, sia stato visualizzato qualcosa, o chiamata una qualsiasi subroutine.
Il programma di caricamento sviluppato per gradi più sotto riposiziona RAMTOP fin dal principio, ma ad ogni modo, se intendete scrivere qualcosa di vostro, non sarà male prendere l'abitudine di effettuare il CLEAR *prima* di fare qualsiasi altra cosa.

Attenzione. Per chiarire con maggiore enfasi quanto detto prima: per rendere "sicura" l'area di memoria posta al di sopra della locazione "indirizzo", usate CLEAR indirizzo-1, *non* semplicemente CLEAR indirizzo.
Il valore posto dopo il comando CLEAR riguarda l'*ultimo indirizzo non sicuro*, non il primo sicuro. È un fastidio, ma piccolo: badate però di rammentarvene.

MEMORIZZAZIONE DEL LINGUAGGIO MACCHINA

Quello che ci serve è un programma (in Basic) che accetti i codici esadecimali, e li memorizzi ordinatamente nei byte posti al di sopra di RAMTOP. La routine che segue fa proprio questo: nel seguito del capitolo vi faremo diverse aggiunte, in modo da renderla ancor più versatile.

```
  10 CLEAR 31999
  20 PRINT "Indirizzo base: 3200
0"
  30 PRINT "Numero di byte dati:
";
  40 INPUT d: PRINT d
  50 LET b=32000
  60 FOR i=0 TO d: POKE b+i,0: N
EXT i
  90 LET a=b+d
 100 DIM h$(2)
 110 PRINT '"Codice HEX:"
 120 INPUT c$: IF c$="s" THEN GO
 TO 200
 140 PRINT c$+" ";
```

```
 150 LET hm=CODE c$(1)-48-39*(c$
(1)>"9")
 160 LET hl=CODE c$(2)-48-39*(c$
(2)>"9")
 170 POKE a,16*hm+hl
 180 LET a=a+1
 190 GO TO 120
 200 POKE a,201
```

Lo scopo della linea 30 è di riservare un certo numero di locazioni a indirizzi posti subito all'inizio della zona del linguaggio macchina, per contenere dei dati: il che spesso è utile. La linea 60 riempie provvisoriamente questi spazi con degli zeri; successivamente, i dati effettivi vi potranno essere inseriti con dei POKE al momento in cui sarà necessario. Le linee 150-160 servono per la conversione da esadecimale a due numeri (decimali) fra 0 e 15 (invece degli originali 0-9, A-F), in modo che 16*hm + hl fornisce l'effettivo valore decimale corrispondente al codice esadecimale. La linea 170 serve ad inserire (POKE) questo valore al corretto indirizzo.
Quando viene impostato in INPUT "s", che non è una cifra esadecimale, essa agisce da "terminatore", segnalando alla macchina che è terminata (STOP) l'introduzione dei codici. La linea 200 assicura che l'ultima istruzione del linguaggio macchina, sia in ogni caso RET (ritorno al Basic), di codice esadecimale C9 = decimale 201. Come mette in evidenza il Manuale, è importante che ogni routine in linguaggio macchina termini con questa istruzione: perciò il nostro programma di caricamento lo fa automaticamente, proprio per il caso che ve ne foste scordati (se così non fosse, niente di male se c'è un RET finale in più.)

ESECUZIONE DI UN PROGRAMMA IN LINGUAGGIO MACCHINA

Per fare eseguire la routine viene usato il comando Basic USR. Per motivi tecnici, USR è una *funzione*, il cui argomento è l'indirizzo a cui comincia il programma in linguaggio macchina, e il cui risultato è il valore che si trova nel registro BC dello Z 80 quando la routine termina. Mentre però elabora questo "calcolo", lo Spectrum manderà in esecuzione il programma in linguaggio macchina. (La cosa può sembrare un tantino perversa, ma è stato fatto così per rendere il linguaggio macchina facilmente accessibile dal Basic.) Se per esempio il linguaggio macchina inizia dall'indirizzo 32000, come abbiamo stabilito di massima, ci serve un comando tipo

LET y = USR 32000

anche se in pratica sarà raro il caso in cui saremo veramente interessati a conoscere il valore finale di y, dopo tutto! Da notare che qualsiasi altro comando sintatticamente corretto che includa USR può servire allo stesso scopo: ad esempio

RANDOMIZE USR 32000

(particolarmente comodo, perché per scrivere RANDOMIZE basta un singolo tasto), ovvero

RESTORE USR 32000

o quanto altro vi venga in mente...
Se avete previsto delle locazioni per i dati in testa all'area linguaggio macchina, dovrete modificare opportunamente il valore 32000 qui indicato, e evitare che i dati che da lì sono presenti vengano considerati parte del linguaggio macchina (codice). Ad ogni modo, ecco le istruzioni Basic che ci permettono di lanciare il programma in linguaggio macchina. È stato congegnato in modo da permettere a questo punto anche altre opzioni – si veda più oltre.

```
 300>CLS : INPUT "Opzione?",o$:
IF o$="r" THEN GO SUB 400
 399 STOP
 400 LET y=USR (b+d): RETURN
```

SALVATAGGIO DI UN PROGRAMMA IN LINGUAGGIO MACCHINA

Il modo per salvare un programma in linguaggio macchina è per il tramite del SAVE di *byte* (si veda il Manuale d'uso dello Spectrum). Supponiamo che la nostra routine risulti lunga 77 byte, incluso il RET finale. Possiamo allora salvarla, sotto il nome "lm" (a mo' di esempio) con il comando diretto

SAVE "lm"CODE 32000, 77

dove 32000 sta per l'indirizzo di partenza (primo byte del linguaggio macchina), e 77 per la lunghezza in byte del programma in linguaggio macchina. In pratica, dato che abbiamo "normalizzato" il collocamen-

to, non ha molta importanza l'effettiva lunghezza del programma in linguaggio macchina, e possiamo addirittura sostituire una volta per tutte 600 al valore 77 come procedura standard. (Verrà utilizzata una maggior lunghezza di nastro, ma in termini veramente insignificanti in pratica.)
Per caricare nuovamente in memoria i byte così memorizzati, la procedura è altrettanto semplice: basterà il comando diretto

LOAD "lm"CODE

anzi, se avete scordato il nome, ma sapete la posizione corretta sul nastro, potrete usare

LOAD""CODE

Possiamo senz'altro includere entrambe queste due possibilità come opzioni operative del nostro programma di caricamento:

```
320>IF o$="s" THEN GO SUB 500
330 IF o$="l" THEN GO SUB 600
500>INPUT "Nome del file per il
SAVE?"'n$
510 IF n$="" THEN LET n$="LM"
520 SAVE n$CODE b,600: RETURN
600 INPUT "Nome del file da car
icare(LOAD)?";'n$
610 LOAD n$CODE : RETURN
```

STAMPA E VERIFICA

Un'aggiunta finale serve per controllare il listato sullo schermo, o anche per stamparlo su carta se disponiamo di una stampante. (Se troveremo degli errori, questi si potranno correggere mediante dei POKE dei valori giusti ai dati indirizzi. Il programma HELPA, dell'appendice 7, fa questo in maniera ancora più comoda, ma per il momento il nostro programma è adeguato.)

```
340>IF o$="z" THEN GO SUB 700
350 IF o$="p" THEN GO SUB 700
700>FOR i=b TO a
710 LET j=PEEK i: LET jm=INT (j
/16): LET jl=j-16*jm
720 LET h$(1)=CHR$ (jm+48+7*(jm
730 LET h$(2)=CHR$ (jl+48+7*(jl
```

```
 700>FOR i=b TO a
 710 LET j=PEEK i: LET jm=INT (j
/16): LET jl=j-16*jm
 720 LET h$(1)=CHR$ (jm+48+7*(jm
>9))
 730 LET h$(2)=CHR$ (jl+48+7*(jl
>9))
 740 IF o$="p" THEN PRINT i;"   "
+h$+"   ";j: GO TO 760
 750 LPRINT i;"   "+h$+"   ";j
 760 NEXT i: RETURN
```

Otterremo un listato sia in decimale che in esadecimale, ordinato per indirizzo, del nostro programma in linguaggio macchina.
Osservate come nell'impostazione originale dei codici esadecimali in INPUT, le a-f venivano stampate sullo schermo in lettere *minuscole*, mentre queste nuove stampe usano le *maiuscole*. Se questo vi disturba, può essere un buon esercizio escogitare un modo per impedirlo.
Risulta però conveniente poter usare minuscole o maiuscole in modo intercambiabile, e così farò anche nel seguito.
Possiamo impiegare questa opzione per controllare il listato prima, o dopo una esecuzione del programma, purché si aggiunga la linea

370 GO TO 300

in modo da poter riselezionare le diverse opzioni. Per fermare si può aggiungere

360 IF o$ = "a" THEN STOP

Così ora disponiamo delle seguenti possibili opzioni:

a STOP
l LOAD
p PRINT (sullo schermo)
r RUN (del programma in linguaggio macchina)
s SAVE
z COPY (alla stampante, per esempio ZX Printer)

Nel seguito di questo libro, faremo l'ipotesi che vi siate predisposti una copia del precedente programma salvata su nastro, pronta per l'impiego sui diversi programmi in linguaggio macchina sparsi per il testo.

CAPITOLO DIECI

Calcoli aritmetici

La chiave per poter eseguire dei semplici calcoli aritmetici sta nell'esistenza delle istruzioni ADD e SUB, entrambe riferentisi al registro A e che possono utilizzare tutti i modi di indirizzamento, salvo il diretto. Per cominciare, vediamo di scrivere un programma per sommare fra loro i due numeri 4 e 7. Possiamo caricarne uno nel registro A, l'altro nel registro B, sommarli, e poi collocare il risultato in qualche posto. Se usiamo il programma di caricamento già messo a punto, specificando che esiste 1 byte per i dati, potremo collocare la somma nella locazione 32000 (7D00). Per prima cosa, vediamo di tradurre il programma nelle sigle mnemoniche per gli opcode:

Porre 4 nel registro A	LD A, 04
Porre 7 nel registro B	LD B, 07
Sommarli assieme (il risultato va nel registro A)	ADD A, B
Memorizzare il risultato in 32000	LD(7D00), A

Cerchiamo ora nell'appendice i relativi opcode in esadecimale: otteniamo il listato in codice esadecimale.

```
LDAA, 04       3E04
LD B, 07       0607
ADD A, B       80
LD(7D00), A    32007D
```

Osservate come il 7D00 venga scritto nell'ordine inverso (007D), come abbiamo già ricordato che è la norma.
E ora procediamo al caricamento in memoria. Caricate prima di tutto il programma di caricamento nello Spectrum, e date RUN. Quando vi viene chiesto il numero di byte, fornite 1. Poi impostate i codici esadecimali, nella forma di doppietti

```
3e 04 06 07 80 32 00 7d
```

(perché il programma caricatore opera con le minuscole), terminando con

```
s
```

che funge da terminatore (e serve anche ad aggiungere in fondo il codice per il cruciale RET, che avevo scordato!).
Quando vi vengono proposte le opzioni, battete r per RUN.
Benissimo, ma che cosa si ottiene? Per vederlo, la cosa migliore è di selezionare l'opzione p, per avere il listato. Ed eccone il risultato

32000	0B	11
32001	3E	62
32002	04	4
32003	06	6
32004	07	7
32005	80	128
32006	32	50
32007	00	0
32008	7D	125
32009	C9	201

Possiamo scorgere il nostro programma, inserito in memoria a partire da 32001 in su, incluso il RET finale, C9. Troviamo anche il risultato, 11, della nostra somma, posizionato entro la locazione 32000, ossia precisamente dove intendevamo che fosse.
Prima di procedere, provate a scrivere analoghi programmi in linguaggio macchina per eseguire queste operazioni:

18 + 66
13 + 17
23 + 6

ricopiando la struttura precedente, ma ovviamente modificando i valori 04 e 07 perché contengano le nuove coppie di numeri e verificate in ogni caso che il risultato corretto si trovi in 32000.

UN USO MIGLIORE DEI DATI

Ottimo, ma non avremo bisogno di scrivere un nuovo programma, come ora suggerito, per ogni nuova coppia di numeri che vogliamo sommare? Oppure sì?
Possiamo modificare i valori 04 e 07 nei numeri (di 1 byte) che vogliamo, m e n per esempio, facendo

POKE 32002,m : POKE 32004, n

Facendo così però mescoliamo programma e dati, perché i due numeri dovrebbero in effetti essere dati. Qui la cosa importa poco; ma in un programma più complesso è sbagliato che il programma si modifichi nel corso dell'esecuzione (si parla di *automodifica* del programma). Primo, perché così non potete reimpiegarlo; e secondo, perché non resta più traccia di quello che esso era all'origine.
Pertanto, sarà pratica migliore prevedere che i due numeri da sommare siano presenti come dati, vengano poi caricati negli opportuni registri nel corso del programma, sommati, e il risultato venga ancora caricato fra i dati.
Riserveremo dunque tre byte per i nostri dati:

32000	primo numero
32001	secondo numero
32002	risultato dell'addizione

che in esadecimale si scrivono 7D00, 7D01 e 7D02. Il programma inizierà da 7D03, e avrà circa questa forma:

Carica il primo numero in A:	LD A,(7D00)
Carica il secondo numero in B:	LD B,(7D01)
Sommali assieme:	ADD A, B
Poni il risultato (totale) in 32002	LD(7D02), A

Eccellente, salvo per il fatto che, se andate a controllare la tabella degli opcode, troverete che non esiste un'istruzione LD B,(nn). Che peccato! C'è tuttavia il modo di ovviare a questo. Uno dei metodi è di usare l'indirizzamento indiretto tramite il registro HL, cosicché invece di LD B,(7D01) scriveremo

Carica HL con il valore dell'indirizzo	LD HL, 7D01
Carica B tramite HL	LD B, (HL)

Il nostro programma assume la nuova forma

```
LD A, (7D00)        3A 00 7D
LD HL, 7D01         21 01 7D
LD B, (HL)          46
ADD A, B            80
LD (7D02), A        32 02 7D
```

Provvedete a caricarlo col solito programma di caricamento, riservando tre byte per i dati: poi fermate il programma con l'opzione STOP, per poter caricare i dati usando

```
POKE 32000, 44 : POKE 32001, 33
```

se – per esempio – si vogliono sommare 44 e 33. Ora date il comando GO TO 300 per rientrare nel programma di caricamento, selezionando poi l'opzione r per RUN. Poi scegliete p per avere la lista. I primi tre valori elencati sono

```
32000     2C     44
32001     21     33
32002     4D     77
```

cui segue il resto del programma. La somma ottenuta, 77, la troviamo alla locazione 32002, come si sperava.
Provate a cambiare i valori che vengono inseriti con i POKE, e a verificare i relativi risultati.
In particolare, provate

```
POKE 32000, 240 : POKE 32001, 100
```

Troverete che il computer è convinto che 240 + 100 = 84: se la cosa vi può sembrare strana, lo è; ma la colpa è vostra, non del computer! L'istruzione ADD non tiene conto del riporto. Provate a controllare in binario:

```
240            1 1 1 1 0 0 0 0
100          + 0 1 1 0 0 1 0 0
               ---------------
     ┌─────────0 1 0 1 0 1 0 0     =     84
     │         ---------------
     ▼         1 1
     1
```

La somma genera un nono bit, che ovviamente non può stare in un byte di 8 bit, e "casca via" dall'estremo superiore; il risultato che viene fornito è inferiore al vero di 256, che è il valore di posizione del nono bit eliminato (infatti 256 + 84 = 340, la somma "esatta"). Non vengono effettuati controlli, né emessi messaggi circa il possibile errore. Quando scrivete in linguaggio macchina siete affidati solo alle vostre forze. Se non prevedete appositi controlli, questi non verranno effettuati per conto vostro.
In pratica, esistono modi per tener conto dei riporti e di questo tipo di *overflow* (superamento dei limiti): si veda più avanti, quando tratteremo dei flag. Per ora non curiamoci della cosa.

SOTTRAZIONE

E ora vedete se siete capaci di modificare il programma in modo da eseguire la *sottrazione* del secondo numero dal primo. Tutto quel che serve in pratica è cambiare da

```
ADD A, B        80
```

in

```
SUB A, B        90
```

e per il resto seguitare come prima. Sperimentate, ed accertatevi che il programma calcola 44 – 3 = 11; e controllate gli effetti che possono derivare dall'overflow (provate a calcolare 17 – 99).

Un sottoinsieme delle istruzioni dello Z 80

Non penso affatto di descrivervi tutte le 694 istruzioni (opcode) dello Z 80 una per una: sarebbe noioso ed anche non necessario (comunque, guardatevi l'appendice 4). Esamineremo invece un "sottoinsieme" che comprende 30 istruzioni caratteristiche (che consentono di creare 240 comandi reali). Purtroppo, non tutte possono impiegare tutti i diversi modi di indirizzamento. Vi fornisco una succinta tavola sinottica che illustra quali modi possono venire usati dalle diverse istruzioni; i relativi opcode sono riportati nell'appendice 6.

opcode / Modo di indirizzamento	LD	ADD ADC SUB SBC AND OR XOR CP	INC DEC SLA SRA SRL	JR JRC JRNC JRZ JRNZ DJNZ	JP	JPZ JPNZ JPC JPNC JPP JPM	LD	ADD ADC SBC	INC DEC PUSH POP
Via registro	LD r, s	ADD A, r	INC r					ADD HL, r	INC r
Immediato	LD r, n	ADD A, n			JP nn	JPZ nn	LD r, nn		
Diretto	LDA, (nn) LD (nn), A						LD HL, (nn) LD (nn), HL		
Indiretto	LD A, (HL) LD (HL), A	ADD A, (HL)	INC (HL)		JP (HL)				
Indicizzato	LD A, (IY + d) LD (IY + d), A	ADD A, (IY + d)	INC (IY + d)	JR d					
	operazioni a 8 bit				operazioni a 16 bit				

Le notazioni impiegate in questa tabella richiedono qualche spiegazione. Alcuni degli opcode non vi saranno familiari, ma li vedremo meglio in seguito. Per il resto, le convenzioni usate sono:
1. Ogni casella riporta un esempio del formato dei vari tipi di istruzione per il particolare modo di indirizzamento. All'opcode ivi illustrato può venire sostituito un altro qualsiasi di quelli citati in testa alla colonna.
2. r o s sta per un generico registro: se si tratti di registro a 8 o a 16 bit dipende dal punto della tabella in cui si trova l'istruzione. Per esempio, nell'istruzione LD r, s, r ed s possono essere uno qualsiasi dei registri di 8 bit (A, B, C, D, E, H, L), mentre in ADD HL, r il registro r è uno fra BC, DE, HK, SP.
3. n sta per un numero di 8 bit; nn per un numero di 16 bit (2 byte).
4. Se viene citato esplicitamente un certo registro, come in LD A,(nn), si tratta dell'unico registro che può venire usato in tale istruzione.
Si tratta però di una regola che può ammettere eccezioni. In alcuni casi si possono impiegare altri registri; ma il fatto è che le istruzioni che vi ho indicato sono tutte corrette, mentre per ampliare il vostro bagaglio di istruzioni potete attendere il momento in cui avrete acquistato un sufficiente grado di confidenza con il sottoinsieme qui specificato.
5. d è un qualsiasi numero di 8 bit, che però viene sommato ad un valore a 16 bit. In altre parole, è il valore di uno spostamento (offset) per l'indirizzamento indicizzato.
Adesso diamo uno sguardo ai nuovi opcode.

AND

Questa operazione considera il contenuto del registro A e quello di un'altro campo di 8 bit, e li raffronta bit per bit: solo se i bit corrispondenti (cioè di posto eguale) sono entrambi 1, pone 1 nella medesima posizione del registro A; altrimenti vi pone uno 0.
Per fare un esempio: AND A, 07 ha questo effetto:

Registro A prima dell'operazione:	0	0	1	1	0	1	0	1
07:	0	0	0	0	0	1	1	1
Registro A dopo l'AND:	0	0	0	0	0	1	0	1

Vedete come sono stati conservati solo i 3 bit inferiori del contenuto originario di A? Ecco quindi un modo di usare AND per "selezionare" una determinata porzione di un byte.

OR

Opera in modo simile ad AND, ma qui la regola è che il risultato di ciascun confronto di bit vale 1 se almeno uno dei due bit confrontati è 1 (vale 0 solo se entrambi sono 0). Così OR A, 05 dà

Registro A prima dell'OR:	0	1	0	0	1	0	1	1
05:	0	0	0	0	0	1	0	1
Registro A dopo l'OR:	0	1	0	0	1	1	1	1

Come vedete, alcuni bit sono stati forzati a 1 qualunque fosse il valore iniziale in quella posizione di A.

XOR

Qui la regola è che si avrà 1 nella posizione per cui i 2 bit corrispondenti sono differenti (uno 0 e l'altro 1). XOR A, B3 dà

Registro A prima:	0	1	0	1	1	0	1	0
B3:	1	0	1	1	0	0	1	1
Registro A dopo:	1	1	1	0	1	0	0	1

Questa istruzione risulta utile per esempio per "commutare" un dato registro fra i valori 0 e 1 alternativamente. Se il registro A contiene inizialmente 0, ogni volta che viene eseguita l'istruzione XOR A, 01 il valore nel registro A "commuta" (da 0 a 1 la prima volta, poi da 1 a 0, da 0 a 1 ecc.).

CP

Questa è l'istruzione di "confronto" (*ComPare*). Il contenuto del registro A viene confrontato con quello di un altro campo di 8 bit. E qui però sorge un problema: come verrà segnalato il risultato del confronto?
A questo scopo viene usato il registro F (o registro *flags*: *flag* significa bandiera in inglese; qui il termine sta a indicare che si "alza" o si "abbassa" una bandierina di segnalazione). Ognuno dei vari bit del registro F contiene una certa informazione circa l'effetto dell'ultima istruzione che li ha modificati. (Non tutte le istruzioni modificano i diversi flag.)
I flag che presentano maggiore interesse per noi sono (si usano universalmente i termini inglesi, e la loro iniziale compare in certe istruzioni): *Carry* (riporto), *Zero, Overflow* e *Sign* (segno).

CP può modificarne uno o più a seconda dei casi, ma il caso che qui ci interessa è la modifica del flag Zero, che viene posto a 1 ("settato" o "posizionato") quando i due valori che vengono confrontati risultano eguali.
Se il contenuto del registro A risulta *inferiore* a quello dell'altro byte con cui viene paragonato, viene posto ad 1 il flag del segno (Sign): un modo di dire "il risultato della sottrazione è negativo" (da notare che la sottrazione non viene realmente effettuata: i valori originali vengono conservati).
Per ora, quanto sopra è sufficiente per quanto riguarda i flag: si tratta di un argomento abbastanza complicato, quando si vuole andare a fondo. Per altre informazioni su questo argomento, si vedano il capitolo 16 e l'appendice 5.

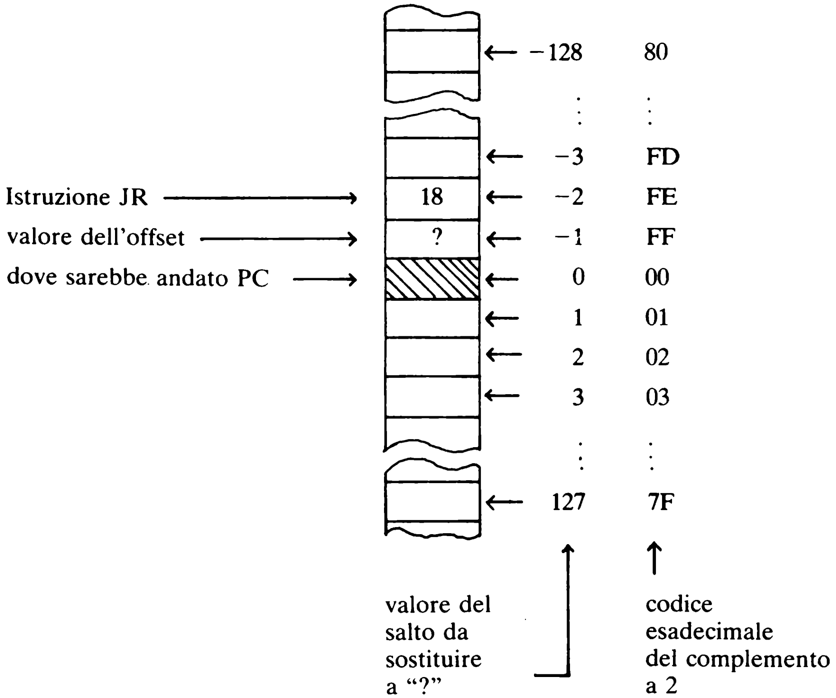
I SALTI

Tutti i salti "condizionati" vengono effettuati (o meno) a seconda del contenuto di vari flag. Così, ad esempio, JPZ significa "salta se il flag Zero è posizionato (a 1)". Ora vediamo come si può usare l'istruzione CP. Supponiamo, per esempio, che si desideri controllare se un dato byte, puntato (indirizzato) da HL, contiene il valore esadecimale 1E. Se è così, vogliamo che si effettui un salto all'indirizzo 447B. Il linguaggio macchina è:

```
LD A, 1E        3E1E
CP A, (HL)      BE
JPZ 447B        CA7B44
```

Anche gli altri tipi di salto hanno comportamenti simili: JPNZ significa "salta se il risultato non è zero" (flag Zero = 0); JPP significa "salta se il risultato è positivo" (flag Sign = 0); JPM significa "salta se il risultato è negativo" (flag Sign = 1); JPNC significa "salta se non c'è riporto" (flag Carry = 0); e così via.
Tutti hanno in comune una cosa, ossia che l'indirizzo dove saltare è fisso. In altre parole se, per un motivo qualsiasi, desideriamo che una certa routine sia collocata ed eseguita da una collocazione in memoria diversa da quella dove è stata caricata originariamente, bisogna provvedere a cambiare tutti gli indirizzi di salto. Lo Z 80 risolve elegantemente questo problema adottando i "salti relativi" (JR). Ossia, il salto viene effettuato spostandosi di tanti byte in avanti (o indietro), a partire dalla posizione in cui ci si trova al momento. Questo spostamento (offset) è contenuto (in notazione con complemento a 2, vedi l'appendice 1) in un singolo byte, cosicché la massima distanza a cui il salto può arrivare è di 128 byte indietro, o 127 byte in avanti.

In pratica lo spostamento viene calcolato dal punto che il PC indicava come indirizzo immediatamente successivo, se *non* ci fosse stato il salto: ovverossia, l'indirizzo della successiva istruzione del programma. Come qui illustrato:



Eccovi un esempio. Vogliamo esaminare in successione ogni byte della memoria alla ricerca del primo valore esadecimale 1E ivi situato. Per semplicità, supponiamo che l'indirizzo di partenza sia già presente in HL. Potremmo scrivere:

```
        LD A, 1E
loop:   CP A,(HL)
        INC HL
        JRNZ loop
```

Due punti necessitano di essere chiariti. Primo, ho infiltrato una nuova istruzione: INC, abbreviazione per INCrementa. Al contenuto del registro specificato viene aggiunto 1: in questo modo, l'operazione di confronto viene ogni volta effettuata con la locazione successiva, perché a ogni ciclo HL viene incrementato di 1. (Per inciso, l'istruzione DEC, per DECrementa, ha l'effetto esattamente opposto a INC.) Il secondo punto è che apparentemente non c'è differenza fra usare JRNZ loop e JPNZ

loop (intendendo che il salto quando avviene va all'istruzione preceduta da loop).
In realtà, la differenza appare chiara dopo che si è proceduto ad assemblare (codificare) le istruzioni in vero linguaggio macchina. Supponiamo che il codice vada caricato a partire dalla locazione 7D00, come al solito:

Indirizzo		Istruzione	Codice esadecimale
7D00		LD A, 1E	3E 1E
7D02	loop:	CP A,(HL)	BE
7D03		INC HL	23
7D04		JRNZ loop	20 FC

Da dove viene quel FC che troviamo nel campo indirizzo dell'istruzione JRNZ? Quando viene eseguita l'istruzione JRNZ, il registro contatore di programma PC viene incrementato di 2, perché si tratta di un'istruzione lunga due byte. Ora dunque il PC contiene 7D06. Noi vogliamo saltare a loop, che si trova all'indirizzo 7D02, 4 byte più indietro, cioè a distanza di –4 posti, per usare il modo di pensare dello Z 80. Ora 4 in binario è 00000100, e per avere –4 dobbiamo invertire i vari bit e sommare 1 (complemento a 2, ricordate?). A questo modo:

```
0 0 0 0 0 1 0 0
                        invertire tutti i bit
1 1 1 1 1 0 1 1
            +   1       sommare 1
---------------
1 1 1 1 | 1 1 0 0
        |               da binario a esadecimale
   F    |    C
```

Per chiarire un altro punto che forse vi preoccupa: INC HL non modifica alcun flag, e di conseguenza sono sicuro per quanto riguarda il test da fare dopo l'incremento (con CP).
Lo stesso programma scritto con salti "assoluti" risulterebbe:

Indirizzo		Istruzione	Codice esadecimale
7D00		LD A, 1E	3E 1E
7D02	loop:	CP A,(HL)	BE
7D03		INC HL	23
7D04		JPNZ loop	C2 02 7D

Osservate come l'istruzione JPNZ sia lunga tre byte, perché contiene un intero indirizzo a 16 bit: e non scordate che i due byte di tale indirizzo vanno scambiati di posto!
Nel gruppo delle istruzioni di salto ce n'è una assai potente, che non vi ho ancora menzionato: si tratta di DJNZ. Essa provvede a decrementare il registro B di 1, e poi salta (in modo relativo) solo se il risultato dell'ultima operazione aritmetica o di confronto è diverso da zero.
Supponiamo che il nostro programmino di "ricerca del valore 1E" debba compiere tale ricerca sopra una zona di soli 100 byte (64 in esadecimale), dopo di che deve uscire dal ciclo sia che la ricerca abbia avuto esito positivo sia in caso contrario:

```
       LD B, 64      06 64
       LD A, 1E      3E 1E
loop:  CP A,(HL)     BE
       JPZ preso     CA xx xx indirizzo esadecimale
                       di "preso")
       INC HL        23
       DJNZ loop     10 F9
```

Il ciclo viene iterato per 100 volte, a meno che prima non venga trovato un byte 1E, nel qual caso ha luogo il salto all'indirizzo "preso". In altre parole, DJNZ funziona in modo assai simile al semplice ciclo iterativo FOR...NEXT del Basic.
Notate come per tutte le istruzioni di salto relativo – JR, JRC, JRNC, JRNZ e JRZ – l'entità del salto venga calcolata allo stesso modo. Una tabella dei valori in complemento a 2 in esadecimale è fornita nell'appendice 1 per la codificazione "a mano" dei salti; il nostro programma di utilità HELPA dell'appendice 7 provvede al calcolo dei valori dei salti relativi in modo automatico, per conto vostro, all'atto della scrittura del codice macchina.

ADC E SBC

Si tratta delle istruzioni di "somma con riporto" (*carry* in inglese) e "sottrazione con riporto". Vi ho detto prima che nel registro flag esiste un bit (flag) del Carry. Esso viene posto a 1 ogni volta che un'istruzione di tipo aritmetico produce un riporto.
L'istruzione ADC funziona esattamente come ADD, salvo che aggiunge 1 se il flag Carry è stato precedentemente posto a 1 da una certa operazione. L'istruzione SBC funziona allo stesso modo, salvo che viene sottratto il valore del flag Carry.

GLI SPOSTAMENTI

Le istruzioni di spostamento (*shift*), SLA, SRA e SRL, hanno tutte l'effetto di spostare i bit di una certa configurazione.
SLA esegue uno spostamento di un posto verso sinistra, così se il registro B contiene

0	0	1	0	1	1	0	0

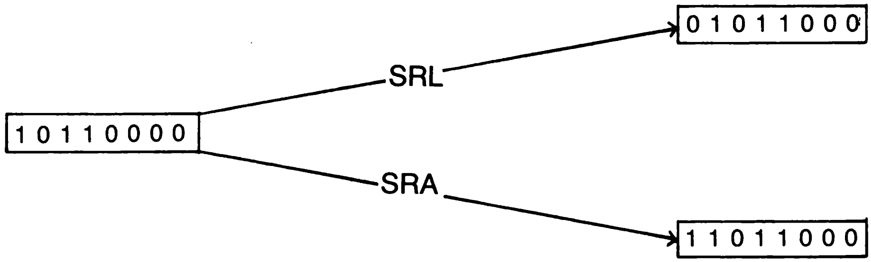
e viene eseguito SLA B, il risultato (in B) è:

0	1	0	1	1	0	0	0

(osservate come venga introdotto uno 0 per completare il byte sulla destra).
Dato che 00101100 = 44 e 01011000 = 88 (in decimale), potete notare che l'effetto generale è quello di moltiplicare per 2.
Un altro SLA B produrrebbe

1	0	1	1	0	0	0	0

Dato che il bit più significativo ("alto": il primo a sinistra) è ora 1, questo numero viene interpretato come negativo, e di conseguenza il flag del segno (Sign) verrà posto a 1. Dal punto di vista del programmatore, in pratica, quello che è successo è che il valore del risultato (176) non può essere contenuto in un byte, pertanto si è verificata una condizione di "*overflow*".
Gli spostamenti verso destra (SRA, SRL) agiscono allo stesso modo, però c'è da notare una cosa importante: SRL provvede a inserire nel bit più a sinistra, per completare il byte, uno zero, in ogni caso; invece SRA vi rimette il valore che c'era prima. Vediamo con un esempio:



La ragione è questa: SRL (*Shift Right Logical*) è uno spostamento verso destra logico, che semplicemente sposta i bit senza alterarli; SRA (*Shift Right Aritmetic*) è un'operazione aritmetica, che produce il risultato di "divisione per 2". Ora, quando un numero negativo viene diviso per 2, il risultato è ancora negativo, e pertanto dobbiamo conservare il bit del segno (il settimo).

PUSH E POP

Forse ricorderete questi termini dalla nostra discussione a proposito dello stack. Essi vengono usati anche qui, esattamente allo stesso modo, e ci permettono di accedere allo stack della macchina, anche quando non viene eseguita una chiamata di subroutine.
La cosa può risultare utile per conservare temporaneamente alcuni valori. Supponiamo ad esempio di avere in BC un valore che desideriamo conservare per più tardi, ma al momento ci serva poter usare il registro BC per qualcos'altro. Si può scrivere:

```
PUSH BC
.....
(linguaggio macchina
che usa
BC)
......
POP BC
```

Il metodo è impiegato spesso anche prima della chiamata (CALL) di una particolare subroutine, di modo che non importi quali registri tale subroutine utilizzi poi per proprio conto: essa non potrà così interferire con i dati del programma chiamante. Possiamo avere del linguaggio macchina scritto in questa maniera:

```
PUSH BC ┐
PUSH DE ├────    salva i tre registri
PUSH HL ┘
```

CALL 4FA1

POP HL
POP DE — recupera i valori originari dei 3 registri (si noti l'ordine in cui avviene il recupero!)
POP BC

(nell'ipotesi che il registro A venga manipolato da parte della routine, cosicché non ci serve salvarne il contenuto).

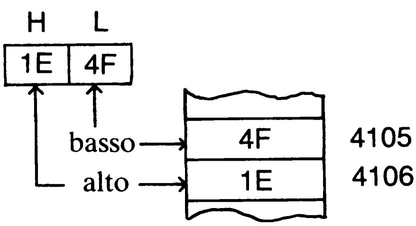
Attenzione! A meno che non scegliate deliberatamente di alterarlo, lo Stack Pointer SP verrà volta a volta fissato secondo le direttive del sistema operativo dello Spectrum. Non succede nulla di male a lasciarlo così, purché prendiate la precauzione di far sì che i PUSH e i POP si bilancino esattamente di numero, in modo che lo SP possa tornare al suo valore originale quando si esce dalla routine in linguaggio macchina. Similmente, anche il numero dei CALL e dei RET deve corrispondere. (Anche USR corrisponde a un CALL, cui corrisponde il RET finale che abbiamo appositamente fatto aggiungere in fondo da parte del programma Basic di caricamento).

UNA PARTICOLARITÀ A 16 BIT

Una caratteristica particolare delle operazioni a 16 bit (riguardante specialmente PUSH, POP e LD), che occorre avere ben compreso, sta nell'ordine con cui vengono trasferiti i byte dai registri alla memoria e viceversa. Le cose vanno a questo modo:

LD(4105), HL

se HL contiene (per esempio) 1E4F, produrrà il seguente effetto:



In altre parole, il byte meno significativo ("basso") del registro viene caricato (per primo) entro la locazione di cui è dato l'indirizzo specifico; mentre il byte più significativo ("alto") viene posto entro il byte successivo in memoria. Inversamente,

LD HL,(4105)

produrrà l'effetto esattamente opposto. (NB: il suo codice esadecimale è 2A 05 41, secondo la solita regola!). Analogamente

LD HL, 1000

(che intende caricare il valore esadecimale 1000 in HL) ha il codice

21 00 10

e così, anche se 1000 è un dato, e non un indirizzo, i relativi 2 byte vengono scritti scambiati di posto.

BLOCCO DI UN PROGRAMMA

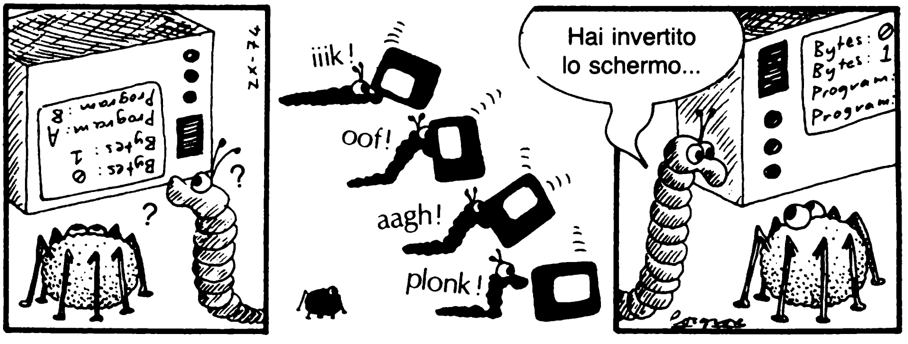
Quando capita che un programma in Basic si blocchi, potete sempre sfuggirne fuori con BREAK, in un modo o nell'altro, senza perdere il programma in memoria. I blocchi che si verificano con il linguaggio macchina sono molto più fastidiosi. Può accadere che producano la cancellazione completa di quanto c'è in memoria (come un NEW), oppure che la macchina resti completamente bloccata, senza rispondere ad alcun tasto, e sia necessario togliere l'alimentazione per qualche secondo – col medesimo risultato. Volete vedere uno di questi blocchi? Eccone un esempio in cui non si ha il reset totale tipo NEW:

RANDOMIZE USR 996

Suoni strani; colori strani; e nessuna risposta della tastiera. Ahimé!
Con RANDOMIZE USR 1400 otterrete un risultato ancora più grigio...
(Il buon vecchio ZX 81 era solito produrre fantastici effetti di "op-art" a seguito di blocchi in linguaggio macchina: lo Spectrum si comporta in maniera molto più "civile", perché è meglio protetto; o alla peggio si riprende la palla e rifiuta semplicemente di giocare oltre.)
Blocchi di questo tipo non si possono mai evitare quando si lavora in linguaggio macchina – come farete presto a imparare da soli, a vostre spese.
L'unico rimedio certo è togliere l'alimentazione: dopo di che dovete ricominciare da capo. Vi sono però alcune precauzioni che val la pena prendere, per ridurre le possibilità di questi guai.

1. Controllate con cura tutti i listati in linguaggio macchina, e accertatevi di averli impostati correttamente in memoria.
2. *Non* usate mai HALT (opcode esadecimale 76): *non* è come il comando Basic di STOP – semplicemente, vi blocca la macchina.

3. Accertatevi che il numero dei vari CALL e RET, nonché dei PUSH e POP, corrisponda.
4. Fate la chiamata al *giusto* indirizzo di partenza del linguaggio macchina.
5. A meno che la perdita sia minima, fate il SAVE su nastro del linguaggio macchina (come byte, come detto) prima di sperimentare per la prima volta l'esecuzione del vostro programma.



CAPITOLO DODICI

Moltiplicazione in linguaggio macchina

Proveremo adesso a scrivere qualche semplice programma. Vi ricordate che vi avevo detto che non esiste un'istruzione dello Z 80 per il prodotto? Vediamo di scrivere una subroutine che possa svolgere questo calcolo.

UN ESEMPIO

Per prima cosa è sempre bene esaminare la natura di un problema, e non c'è modo migliore che operare con un esempio. Cercheremo di mantenere le cose semplici, e di lavorare perciò con registri di 8 bit: la moltiplicazione di 9 × 13 in binario si presenterà così

```
  0 0 0 0 1 0 0 1

× 0 0 0 0 1 1 0 1
```

Potremmo operare come si fa di solito per una moltiplicazione con numeri (decimali) lunghi ma, dato che siamo nel sistema binario, la cosa risulta molto semplificata: se la cifra del moltiplicatore che stiamo considerando volta a volta è 1, si copia il moltiplicando, se invece è zero, non si fa nulla, e si passa alla cifra seguente:

```
  0 0 0 0 1 0 0 1   P
x 0 0 0 0 1 1 0 1   Q
-----------------
  0 0 0 0 1 0 0 1
  0 0 1 0 0 1 0 0
  0 1 0 0 1 0 0 0
-----------------
  0 1 1 1 0 1 0 1
```

Naturalmente, abbiamo dovuto aggiungere degli 0 sulla destra, come faremmo in una moltiplicazione decimale (dove in effetti non li scriviamo neppure). In linguaggio macchina, l'abbiamo già visto, questo corrisponde a uno spostamento verso sinistra (SLA). Per comodità, ho chiamato i due numeri *P* e *Q*.

Mentre si esegue lo spostamento a sinistra di *P*, risulta utile effettuare anche un successivo spostamento di *Q* verso destra, perché in questo modo ci basta esaminare a ogni passo soltanto il bit più basso (a destra) di *Q* per stabilire se *P* va sommato oppure no.

Procedimento

Poniamo che *P* e *Q* si trovino rispettivamente nei registri D ed E.
Il procedimento da seguire è il seguente:
1. azzerare il registro A;
2. se il bit meno significativo (numero 0) di E vale 1, sommare D ad A;
3. spostare D a sinistra di un bit;
4. spostare E a destra di un bit;

I passi da 2 a 4 vanno ripetuti per 8 volte.
Proviamo a stendere il corrispondente codice in linguaggio macchina:

```
LD A, 00
LD B, 08
```

La prima istruzione è ovvia; la seconda si è adottata per usare B come contatore, in collegamento con un'istruzione DJNZ da porre verso la fine. Adesso si tratta di esaminare il bit "basso" di E.
L'unico modo che attualmente conosciamo è usare una "maschera", cioè una opportuna configurazione di bit (00000001) assieme a un operatore AND; poniamo dunque tale "maschera" di bit in C:

```
LD C, 01      (per i codici esadecimali, vedi oltre).
```

L'operazione AND si può effettuare solo sul registro A, ma allora

l'attuale contenuto andrebbe perso, e quindi lo salviamo prima nel registro L:

```
loop: LD L, A
```

Poi estraiamo il bit più basso di E, e ripristiniamo il contenuto del registro A:

```
LD A, C
AND A, E
LD A, L
```

Se il risultato dell'AND era zero, dobbiamo saltare la parte "sommare D ad A" del passo 2, e pertanto

```
JRZ shift
```

(Notare che siccome LD non modifica i flag, JRZ si riferisce ancora all'ultima operazione – AND – che ha potuto modificarli.) In caso contrario, va eseguita la somma:

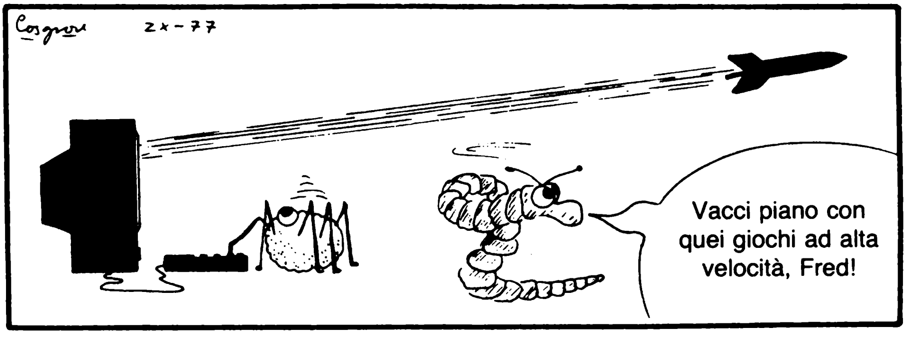
```
ADD A, D
```

E ora effettuiamo gli spostamenti:

```
shift:  SLA D
        SRA E
```

e verifichiamo se abbiamo completato il prescritto numero di cicli:

```
DJNZ loop
(RET)
```



Il codice in linguaggio macchina

Ed ecco l'intero programmino:

```
       LD A, 00       3E 00
       LD B, 08       06 08
       LD C, 01       0E 01
loop:  LD L, A        6F
       LD A, C        79
       AND A, E       A3
       LD A, L        7D
       JRZ shift      28 01
       ADD A, D       82
shift: SLA D          CB 22
       SRA E          CB 2B
       DJNZ loop      10 FE
```

Se volete provare il funzionamento del programma, dovete provvedere a collocare in D ed E i numeri che si vogliono moltiplicare.
Conviene allora far precedere il programma da qualcosa come:

```
LD HL, 7D00     21 00 7D
LD D, (HL)      56
INC HL          23
LD E, (HL)      5E
```

e quindi eseguire il POKE 7D00 (esadecimale) e POKE 7D01 (esadecimale) dei valori dei due fattori, prima di lanciare il programma. Questi due byte saranno, naturalmente, i due byte contenenti 0 posti prima dell'inizio della routine in memoria, per cui l'istruzione LD HL, 7D00 avrà l'indirizzo 7D02. Notate come non abbia assegnato specifici indirizzi dentro al programma: questo perché tutti i salti sono di tipo relativo, e non hanno importanza gli indirizzi effettivi; contano solo gli offset.
Bisogna inoltre "portar fuori" in qualche modo il risultato: per il momento, esso si limita a starsene nel registro A. Una maniera è di riservare un'altra locazione dati, 7D02, per la risposta, così come fatto nel precedente programma di somma: ciò significa che si dovrà aggiungere in fondo al codice

```
LD(7D02), A        32 02 7D
```

e usare il valore 3 come risposta al numero di dati chiesti dal programma di caricamento. Per ricavare il risultato, si userà

```
PRINT PEEK 32002.
```

Altrimenti, possiamo trasferire il risultato dal registro A al registro C, impiegando le istruzioni

```
LD B, 00        06 00
LD C, A         4F
```

ed effettuando la chiamata del linguaggio macchina (che in questo caso comincia sempre da 7D02) con

```
PRINT USR 32002
```

Ricordate che USR è una funzione, che al momento del ritorno al Basic contiene il valore contenuto nel registro BC: in questo modo tale istruzione a un tempo lancia il programma in linguaggio macchina e poi stampa il risultato! (Come detto, siamo ancora nel caso di due soli byte per i dati, da cui il 32002 = 7D02 come argomento di USR.)

BIT

A questo punto, devo fare una confessione: in effetti, esiste un modo più facile per controllare se il bit inferiore di E contiene 1. Esiste infatti un'istruzione BIT 0, E che compie il medesimo lavoro. In tal modo la parte di programma che consiste in

```
loop:  LD L, A         6F
       LD A, C         79
       AND A, E        A3
```

viene semplicemente sostituita con

```
loop:  BIT 0, E       CB 43
```

e non serve più nemmeno l'istruzione LD A, L.

Perché non ve l'ho detto subito? Beh, in primo luogo, avevo promesso di utilizzare solo le istruzioni della tabella del capitolo 11, promessa che ora ho rotto. Però in questo modo ho evidenziato un punto importante: e cioè che, anche senza conoscere il gruppo completo delle istruzioni, si

possono raggiungere soddisfacentemente i propri scopi anche con mezzi più limitati.
L'esempio fornito è un po' accademico: l'ho scelto perché in esso vengono impiegate diverse istruzioni di tipo comune in modo convenzionale, ma non necessariamente ovvio: non voglio dare però l'idea che troverete comunemente necessità di calcolare dozzine di prodotti fra numeri di 8 bit!

CAPITOLO TREDICI

Lo schermo

Le informazioni che controllano l'immagine sullo schermo sono contenute in due sezioni della RAM, note come la *memoria dello schermo* e la *memoria degli attributi*. La prima controlla i caratteri e la grafica ad alta risoluzione, mentre la seconda controlla i colori e gli altri attributi quali FLASH (lampeggio) e BRIGHT (luminosità).
Esse sono ordinate in modo alquanto diverso. La più semplice risulta la memoria degli attributi, pertanto inizierò da questa.

LA MEMORIA DEGLI ATTRIBUTI

Essa occupa 24 × 32 = 768 byte posti nelle seguenti locazioni:

	Decimale	Esadecimale
Inizio della memoria attributi	22528	5800
Fine della memoria attributi	23295	5AFF

I primi 22 × 32 = 704 byte servono a manipolare la normale area di PRINT sullo schermo, vale a dire le righe da 0 a 21 dove si può usare PRINT AT. Gli ultimi 2 × 32 = 64 riguardano invece la zona delle due righe di fondo dello schermo, solitamente utilizzata per le modifiche alle istruzioni (EDITing), gli INPUT ed i messaggi di errore. La regione dove si può normalmente effettuare il PRINT termina all'indirizzo decimale

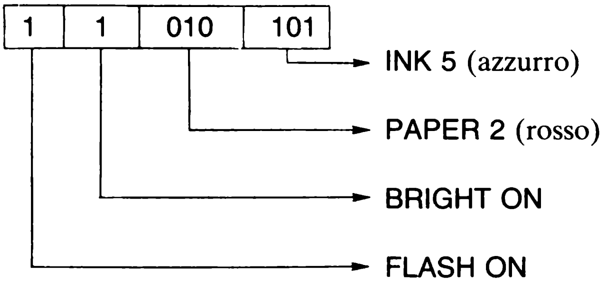
23231 (5ABF esadecimale); quindi la parte inferiore dello schermo inizia da 23232 (5ACO esadecimale).
Se numeriamo come di consueto le righe da 0 a 21 (e pensiamo alle due righe in fondo come righe 22 e 23), e le colonne da 0 a 31, allora la posizione posta alla riga *r* e colonna *c* ha per indirizzo

$22528 + 32 \times r + c$

ed è a questo indirizzo che è contenuto il valore degli *attributi* di quella posizione.
Ogni attributo è costituito da un singolo byte, i cui 8 bit sono suddivisi nel modo seguente:

FLASH on/off	BRIGHT on/off	tre bit per colore PAPER	tre bit per colore INK

Nel caso particolare dei bit relativi a FLASH e BRIGHT, "1" significa ON e "0" significa OFF. Se il byte ha la struttura



gli attributi saranno come indicato (una combinazione alquanto orribile, per inciso...). Se si vuole che questa combinazione di attributi corrisponda al "quadratino" dello schermo posto alla riga 5, colonna 7, bisognerà memorizzare tale byte nella locazione

$22528 + 32 \times 5 + 7$

corrispondente a

22695.

Il valore di quel byte è 213 in decimale, pertanto si farà

POKE 22695, 213

per memorizzare il byte al posto giusto. Ovviamente non vedrete il colore di INK se quel che è visualizzato è solo uno spazio vuoto, quindi bisognerà stampare qualcosa in quella posizione, per esempio con

```
PRINT AT 5,7;"*"
```

e così potrete accertarvi che la cosa funziona.
Dato che in ogni riga ci sono 32 colonne, l'indirizzo di una posizione posta subito sotto un'altra si trova sommando 32 decimale (20 esadecimale) all'indirizzo della precedente. La memoria degli attributi ha dunque in sostanza questa struttura:

riga di testa	5800	5801	5802	...	581D	581E	581F	area di stampa
prima riga	5820	5821	5822	...	583D	583E	583F	
seconda riga	5840	5841	5842	...	585D	585E	585F	
.	.	.	.	...	.	.	.	
.	.	.	.	...	.	.	.	
.	.	.	.	...	.	.	.	
21-esima riga	5AA0	5AA1	5AA2	...	5ABD	5ABE	5ABF	
22-esima riga	5AC0	5AC1	5AC2	...	5ADD	5ADE	5ADF	messaggi
23-esima riga	5AE0	5AE1	5AE2	...	5AFD	5AFE	5AFF	

LA MEMORIA DELLO SCHERMO

Questa risulta molto più complicata come struttura, perché ciascun carattere è memorizzato come gruppo di *otto* byte, ognuno dei quali corrisponde a una riga della matrice di 8 × 8 pixel in alta risoluzione occupati da quel carattere. E per rendere le cose ancora più difficili, essi *non* sono memorizzati nell'ordine che ci si attenderebbe.
Quando "caricate" nello Spectrum un'immagine usando LOAD... SCREEN$, dovreste aver notato la maniera alquanto curiosa con cui viene "dipinto" il quadro. Essa corrisponde esattamente all'ordine con cui i vari byte sono collocati nella memoria dello schermo. Il modo più semplice per controllarlo è riempire lo schermo completamente di nero, poi salvare (SAVE) il quadro e ricaricarlo (LOAD):

```
10 FOR r = 0 TO 21
20 PRINT " [32 spazi inversi] "
30 NEXT r
40 SAVE "nero" SCREEN$
```

(Per inciso: lo spazio inverso si ottiene in "modo grafico" (CAPS SHIFT + 9) con CAPS SHIFT + 8.) Salvate lo schermo su nastro, dopo il RUN, rispondendo all'istruzione 40; poi impostate sulla tastiera

```
CLS:LOAD "nero"SCREEN$
```

e osservate attentamente il modo con cui lo schermo diventa tutto nero. Passiamo ai numeri. Gli indirizzi di questa area sono:

	Decimale	Esadecimale
Inizio della memoria schermo	16384	4000
Termine della memoria schermo	22527	57FF

e il numero totale di byte è di 32 × 24 × 8 = 6144 decimale, 1800 esadecimale.
Il modo migliore di considerare la memoria dello schermo è di immaginare il quadro suddiviso in tre blocchi:

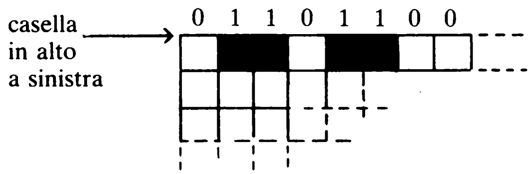
blocco 1 righe 0-7 dello schermo
blocco 2 righe 8-15 dello schermo
blocco 3 righe 16-23 dello schermo

Osservate come il blocco 3 includa anche le due righe destinate agli INPUT e ai messaggi di errore (righe 22 e 23). Ogni blocco è costituito da 6144/3 = 2048 byte (800 in esadecimale). I primi 2048 byte della memoria dello schermo corrispondono al blocco 1, gli ulteriori 2048 al blocco 2, e gli ultimi 2048 al blocco 3; e l'ordinamento dei vari byte è lo stesso nei vari blocchi. In esadecimale abbiamo

Blocco	Indirizzo iniziale	Indirizzo finale
1	4000	47FF
2	4800	4FFF
3	5000	57FF

Per comprendere la disposizione dei byte entro un blocco, considererò il blocco 1: gli altri, come detto, sono del tutto simili (ricordate però che il blocco 3 include le due righe di fondo per i "messaggi di errore"). Il processo si può descrivere meglio facendo riferimento alla griglia di coordinate in alta risoluzione di 256 × 176 pixel.
Numeriamo le "righe" da 0 a 175 e le "colonne" da 0 a 255, con la riga 175 in cima e la colonna 0 all'estremo sinistro, come avviene normal-

mente per le indicazioni relative alle istruzioni PLOT *colonna*, *riga*. Il blocco 1 comprende le righe 175-112, per un totale di 8 × 8 = 64 righe, ciascuna di 256 pixel.
I primi 32 byte della memoria schermo contengono le informazioni relative alla riga 175. Ogni byte corrisponde e gestisce un segmento di 8 colonne successive. Il primo byte è relativo alle colonne 0-8, usando 1 bit per pixel. Ad esempio, se l'indirizzo 4000 (esadecimale) contiene il byte 01101100 = 108, allora i primi 8 pixel in cima a sinistra nello schermo avranno il seguente aspetto:



dove troviamo "spento" per 0 e "illuminato" per 1.
Il byte successivo è relativo alle colonne 8-15, il successivo alle colonne 16-23, e così via fino a che sono stati compresi tutti i 32 byte della riga 175.
Per vedere il tutto in funzione, fate CLS e poi

```
POKE 16384, BIN 01101100
```

e vedrete comparire due corti tratti (corrispondenti alle parti 11) in cima allo schermo. Provate anche con

```
POKE 16384, BIN 10101010
```

e otterrete 4 punti "neri". Ora modificate 16384 in 16385, 16386, ... e via di seguito sino a 16416, e riempirete in tal modo la riga 175.
Il byte successivo è all'indirizzo 16146. Provate

```
POKE 16146, BIN 10101010
```

e vedrete (con sorpresa) che *non* si tratta delle prime 8 colonne della riga successiva (174)! Infatti, i punti compaiono in testa alla riga 167. Osservate come 167 = 175 – 8: ogni volta vengono saltate *otto* righe in alta risoluzione. Questo byte, e i 31 seguenti, riempiono la riga 167. Poi la macchina passa alla riga 167 – 8 = 159, e continua a saltare 8 righe sino a quando arriva al termine del blocco 1. Quindi, le righe, ciascuna di 32 byte, si succedono nell'ordine

175 167 159 151 143 135 127 119

Se sottraiamo ancora 8, abbiamo il numero di riga 111, che non appartiene più al blocco 1. (Infatti si tratta della prima riga del blocco 2). Ecco perché lo Spectrum salta a riempire le varie righe che si trovano immediatamente sotto a quelle che ha appena finito di trattare: le prossime otto sezioni di 32 byte riguardano quindi le righe

174 166 158 150 142 134 126 118

Terminate queste, si passa al gruppo di righe

173 165 157 149 142 132 125 117

e così via, fin quando non si raggiunge la fine del blocco:

168 160 152 144 136 128 120 112

E così si è completato il blocco 1. I successivi 2048 byte corrispondono al blocco 2, ordinati nello stesso modo; infine si arriva al blocco 3. L'ordine dei byte entro ciascun blocco è esattamente lo stesso: gli indirizzi in ciascun blocco differiscono di 2048, a cui corrisponde uno spostamento in basso di 64 righe dello schermo, passando da un blocco al successivo. La cosa può sembrare alquanto complicata, e la cosa migliore è considerare prima i diversi blocchi distintamente, poi questi in spezzoni di 8 righe, e solo poi i vari byte entro ciascuna riga. L'ordinamento sembra più ragionevole se ricordiamo che ogni carattere occupa un quadratino di 8 × 8 pixel (in alta risoluzione); ossia una posizione **PRINT AT**. Allora, le 8 linee di ciascun carattere corrispondono a 8 byte (esattamente allo stesso modo che viene impiegato per definire gli "*User Defined Graphics* (UDG)".

Pertanto, una certa serie di caratteri visualizzati nelle posizioni che corrispondono al blocco 1 (righe 0-7 in bassa risoluzione) sono disposti nella memoria dello schermo a questo modo:

32 byte per la prima linea di ciascun carattere presente nella riga 0
32 byte per la prima linea di ciascun carattere presente nella riga 1
....
32 byte per la prima linea di ciascun carattere presente nella riga 7

e poi

32 byte per la seconda linea di ciascun carattere presente nella riga 0
32 byte per la seconda linea di ciascun carattere presente nella riga 1
....
32 byte per la seconda linea di ciascun carattere presente nella riga 7

e così avanti per la terza, quarta, ..., ottava linea di ciascun carattere. Può sembrare alquanto confuso; però forse a questo punto lo si capisce un po' meglio che all'inizio.
Nel prossimo capitolo prenderemo confidenza con la memoria degli attributi; poi passeremo a quella dello schermo nel capitolo successivo.

CAPITOLO QUATTORDICI

La memoria degli attributi

Il modo più semplice per imparare come usare la memoria degli attributi è di scrivere una routine in linguaggio macchina che serva a proiettare un quadratino di colore assegnato nell'angolo in alto a sinistra (dove ha inizio la zona di memoria, all'indirizzo 5800 esadecimale). Userò come di solito un byte dati per contenere il valore del colore (byte degli attributi); il codice è allora:

```
LD A, attributo      3A 00 7D
LD(5800), A          32 00 58
```

Caricate il programmino, date "s" (STOP), poi inserite il dato con

POKE 32000, 32

(Dato che $32 = 4 \times 8$, questo corrisponde all'attributo PAPER 4). Ora date GO TO 300 e RUN con la "r"... Bene, ha funzionato! Ed eccovi qualche altro esempio da sperimentare: se non ci riuscite da soli, trovate le risposte in fondo al capitolo.

1. Rendete il quadratino rosso anziché verde.
2. Fate diventare violetto il colore del quadratino.
3. Ora fatelo di colore blu, ma lampeggiante (FLASH).
4. E ora giallo e BRIGHT.
5. Spostate il quadratino nella riga 0 alla colonna 1.
6. E ora spostatelo nella riga 3, colonna 7.

Per eseguire gli esempi da 1 a 4 basterà fare POKE 32000 con il valore idoneo; ma per gli esempi 5 e 6 dovrete modificare l'indirizzo 5800 sostituendovi quello della corretta posizione all'interno della memoria degli attributi.
Ora proviamo a colorare l'intera prima riga dello schermo in verde. Ci serve un ciclo, e l'uso di un contatore fissato inizialmente a 32 e decrementato a ogni passo: si verifica se ha raggiunto lo zero, e in caso contrario si salta da capo.

```
        LD HL, a-file        21 00 58
        LD B, 32 dec         06 20
loop:   LD (HL), 32 dec      36 20
        INC HL               23
        DEC B                05
        CP B                 B8
        JRNZ loop            20 F9
```

Questa volta non vi sono byte per i dati, e si passerà direttamente all'opzione "r" quando compare la selezione.
Possiamo facilmente colorare allo stesso modo due, tre, ... righe aumentando il valore del contatore in B. Se iniziamo con LD B, 64(decimale) (06 40) otteniamo *due* righe colorate in verde; (06 60) ne crea tre; e così via sino a (06 E0), che ne dà sette. Se aggiungiamo ancora 20 esadecimale (32 decimale) a B lo poniamo a 00; il primo decremento quindi porta tale valore a 255 (ricordate, non ci sono riporti!), ed è come se si fosse partiti con 256 in B. Infatti, se cambiate la seconda linea del codice qui sopra in

```
LD B, 0              06 00
```

vedrete colorarsi di verde le prime *otto* righe dello schermo. Oltre a questo non si può andare, almeno non semplicemente modificando il valore caricato in B, dato che questo è un registro di 1 byte.
Prima di vedere come si può aggirare il problema, vediamo come si può essere un po' più efficienti. Se si usa l'istruzione DJNZ, l'impiego del registro B come contatore diventa *automatico*, e così è possibile risparmiare un certo numero di byte. Lo stesso lavoro si può quindi ottenere altrettanto bene con

```
         LD HL, a-file        21 00 58
         LD B, 0              06 00
loop:    LD (HL), 32 dec      36 20
         INC HL               23
         DJNZ loop            10 FB
```

Provate.

COLORAZIONE ISTANTANEA DELL'INTERO SCHERMO

Ora, per ottenere il cambiamento di colore di tutte e 24 le righe di cui è composto lo schermo, possiamo usare BC come contatore (facendolo partire da 0300 esadecimale), aggiungendo un CP C e un altro JRNZ *loop* (infatti, per verificare che BC sia diventato zero occorre controllare B e C separatamente). Oppure, possiamo iterare il processo già visto usando un altro registro come contatore. O ancora, da persone assai poco raffinate quali siamo, possiamo semplicemente appiccicare assieme tre copie del programma in serie, partendo ogni volta da diversi indirizzi della memoria degli attributi.
Proviamo prima il caso meno raffinato:

```
         LD HL, 5800          21 00 58
         LD B, 0              06 00
loop 1:  LD (HL), 32 dec      36 20
         INC HL               23
         DJNZ loop 1          10 FB
         LD HL, 5900          21 00 59
         LD B, 0              06 00
loop 2:  LD (HL), 32 dec      36 20
         INC HL               23
         DJNZ loop 2          10 FB
         LD HL, 5A00          21 00 5A
         LD B, 0              06 00
loop 3:  LD (HL), 32 dec      36 20
         INC HL               23
         DJNZ loop 3          10 FB
```

Date il via a questo programma (opzione "r"), dopo aver indicato 0 byte dati: vedrete lo schermo colorarsi istantaneamente, inclusa la parte

bassa riservata ai "messaggi di errore". Se non volete questo, modificate il terzo LD B, 0 in LD B, 192 (decimale) ovvero (06 C0) per colorare solo 6 righe nell'ultimo passaggio.
Incidentalmente, avete notato come non sia necessario riportare B a zero ogni volta? Infatti ha già questo valore! Possiamo quindi togliere le istruzioni LD B, 0 (tranne ovviamente la prima, che è sempre necessaria; e la terza, se volete solo 22 righe colorate in verde).
Ovvio che deve esistere una maniera più rapida per ottenere lo stesso effetto; comunque quella indicata sopra ha una caratteristica interessante: possiamo modificare il colore di ciascun blocco separatamente. Se modificate il secondo (36 20) in (36 10) e il terzo in (36 30) (il modo più semplice è di fare POKE 32016,16: POKE 32026,48), otterrete tre blocchi colorati sullo schermo:

VERDE
ROSSO
GIALLO

Naturalmente, se vengono modificati i byte degli attributi, si ottengono altri effetti simili.
Non ci sono scuse: vediamo di far funzionare in modo appropriato quell'ulteriore ciclo cui abbiamo accennato prima. Useremo il registro D come contatore del ciclo. Tanto per cambiare, stavolta vogliamo uno schermo tutto porpora

```
        LD HL, a-file       21 00 58
        LD D, 03            16 03
outer:  LD B, 0             06 00
loop:   LD (HL), 24 dec     36 18
        INC HL              23
        DJNZ loop           10 FB
        DEC D               15
        CP D                BA
        JRNZ outer          20 F5
```

Notate come non si proceda a incrementare HL entro il ciclo esterno, perché non ce n'è bisogno; esso infatti continua allegramente il suo cammino attraverso la memoria degli attributi da sè: dobbiamo solo rimediare al difetto che B da solo non può servire qui da contatore.

LAMPEGGIO E NO

Il prossimo programma scorre attraverso la memoria degli attributi eliminando il lampeggio (FLASH) in tutti quei quadratini in cui era in atto, lasciando inalterati gli altri attributi. Ora, come sappiamo, il bit del FLASH è quello all'estrema sinistra (bit numero 7). Dobbiamo semplicemente porlo a zero lasciando gli altri bit immutati.
Una delle maniere per ottenerlo è tramite una *maschera* di bit: se il byte degli attributi viene collocato nel registro A e si esegue poi l'AND con la configurazione di bit 01111111, il bit di sinistra passerà comunque a 0, mentre gli altri non verranno modificati.
Impiegando quest'idea, ecco il codice in linguaggio macchina atto allo scopo:

```
       LD BC, 768 dec     01 00 03
       LD HL, a-file      21 00 58
loop:  LD A, (HL)         7E
       AND 127 dec        E6 7F
       LD (HL), A         77
       INC HL             23
       DEC BC             0B
       LD A, 0            3E 00
       CP B               B8
       JRNZ loop          20 F5
       CP C               B9
       JRNZ loop          20 F2
```

Osservate come si utilizzi il registro BC come contatore: come è ovvio, 768 è la lunghezza della memoria degli attributi. Occorre controllare che tanto B quanto C siano 0, per chiudere il ciclo.
Provvedete al caricamento di questa routine, con 0 byte dati; poi rispondete "s" (STOP). Per fare l'esperimento, occorre che ci sia un'immagine sullo schermo. Creiamone una con le seguenti righe aggiuntive di Basic:

```
1000 CLS: FOR i = 1 TO 704: PRINT FLASH
  (INT(2*RND);CHRS(65+i–20*INT(i/20));:
  NEXT i
1020 GO TO 300
```

Per lanciarlo, usate GO TO 1000. Lo schermo si riempirà di lettere, alcune delle quali lampeggianti. Ora rispondete alla selezione delle

opzioni con "r", e vedrete come tutti i caratteri lampeggianti cesseranno di lampeggiare.
Provate a fissare **INK** e **PAPER** di diversi colori, e ripetete: vedrete che i colori non vengono modificati.
Il problema inverso consiste nel rendere lampeggiante l'intero schermo. Invece di usare **AND 127**(decimale) per "mascherare" i bit che interessano, si userà **OR 128**(decimale) (dato che 128 è 10000000 in binario, l'**OR** forza il passaggio a 1 del bit numero 7 lasciando immutati gli altri). Il programma resta lo stesso, salvo per la sostituzione della linea

```
AND 127         E6 7F
```

con la

```
OR 128          F6 80
```

Come prima, **GO TO 1000** per fare la verifica. Che cosa succede con **XOR 128**?

SET E RES

Esiste anche un modo ancora più diretto per modificare la configurazione di bit nel modo desiderato. L'istruzione **SET** serve a *posizionare* (*set*) ad 1 un dato bit, mentre **RES** lo *riposiziona* (*reset*) a 0.
I bit entro un dato byte sono numerati così

7	6	5	4	3	2	1	0

con il numero d'ordine massimo per il bit più a sinistra o "alto". Una istruzione tipo

```
SET 4, D
```

pone a 1 il bit 4 del registro **D**; mentre

```
RES 6, E
```

mette a 0 il bit 6 del registro **E**.
Quindi, invece di usare **AND 127** si sarebbe potuto usare semplicemente

```
RES 7, A        CB BF
```

e invece di OR 128(decimale)

```
SET 7, A            CB FF
```

Queste istruzioni hanno la stessa lunghezza di quelle che sostituiscono, pertanto non c'è bisogno di aggiustare gli offset dei salti relativi (cosa che in altri casi potrebbe essere necessaria). Provate a fare la sostituzione e a verificare che il programma seguiti a funzionare allo stesso modo.

"SCROLLING" DI UNA COLONNA

E ora proviamo a scrivere una routine che esegua lo *scrolling* (spostamento verso l'alto) di una colonna di attributi; per ora, ci fissiamo sulla colonna 31, quella più a destra sullo schermo. Ricordatevi che gli attributi di una casella posta immediatamente sotto una casella data hanno un indirizzo più alto di 32 rispetto a quello della prima. Possiamo quindi usare l'indicizzazione degli indirizzi, con uno spostamento di 32. L'idea da concretare è di trasferire gli attributi della casella inferiore nel registro D, e poi trasferirli nella casella superiore, iterando 21 volte per l'intera colonna. E così si arriva a questo codice:

```
        LD BC, 32 dec       01 20 00
        LD IX, start        DD 21 1F 58
        LD A, 21 dec        3E 15
loop:   LD D, (IX + 32)     DD 56 20
        LD (IX), D          DD 72 00
        ADD IX, BC          DD 09
        DEC A               3D
        CP B                B8
        JRNZ loop           20 F4
```

Osservate come il valore 1F nella seconda istruzione è il numero di colonna, 31, che intendiamo sottoporre allo scrolling, però scritto in esadecimale. Se la colonna fosse diversa, basterà cambiare questo valore in corrispondenza.
Anche stavolta ci occorre un'immagine già fatta con cui sperimentare il programma. Caricate il linguaggio macchina, STOP e poi aggiungete le linee di Basic

```
1000 FOR c = 0 TO 21: PRINT PAPER c - 7*INT(c/7);
  " [32 spazi] ";: NEXT c
1010 GO TO 300
```

Poi GO TO 1000, ottenendo un simpatico quadro a strisce su cui fare la verifica; e quindi scegliete l'opzione "r" per vedere l'effetto.
Vedrete la colonna 31 che si sposta in alto di uno spazio. Per ottenere uno spostamento maggiore, fate iterare la routine a partire da un programmino Basic. Per "scrollare" a partire ad un'altra colonna, cambiate l'1F (come detto prima).
Avrete anche osservato che l'attributo in fondo alla colonna non viene modificato. Per renderlo un quadratino bianco (attributo 56 = 7 × 8) aggiungere la seguente linea al linguaggio macchina, subito dopo JRNZ:

```
LD(IX), 56 decimale         DD 36 00 38
```

Come esercizio: scrivete un programma che realizzi lo scrolling di un intero blocco di colonne (diciamo dalla 5 alla 17 incluse), iterando la routine descritta, in linguaggio macchina, e incrementando ogni volta l'indirizzo iniziale per IX (*start*).

"TAPPETO MAGICO"

Infine, eccovi un programma che, basandosi sulle idee della "colorazione istantanea", fornisce un suggestivo effetto creando "magiche" configurazioni. Lascio a voi scoprire come funziona: vi ricordo soltanto che i byte per i dati sono 0.

```
LD B, 0          06 00
LD D, 0          16 00
LD HL, 5800      21 00 58
LD (HL), D       72
INC HL           23
LD A, 7          3E 07
ADD A, D         82
LD D, A          57
DJNZ ?           10 F8
LD HL, 5900      21 00 59
LD (HL), D       72
INC HL           23
LD A, 7          3E 07
ADD A, D         82
LD D, A          57
DJNZ ?           10 F8
```

LD HL, 5A00	21 00 5A
LD (HL), D	72
INC HL	23
LD A, 7	3E 07
ADD A, D	82
LD D, A	57
DJNZ ?	10 F8

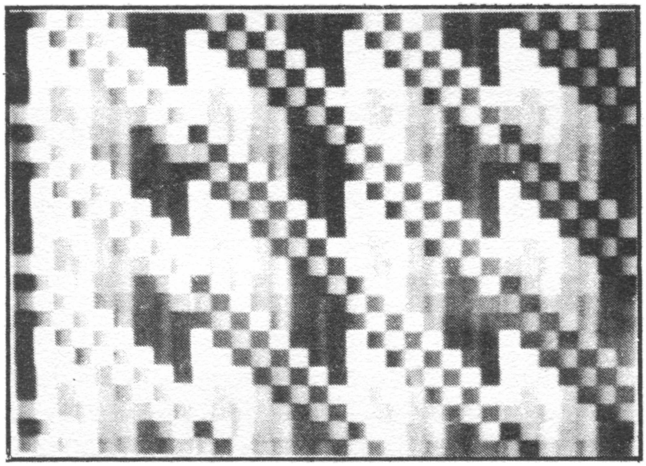


FIG. 14.1. Usate LD A, 31 per questo schema a scacchiera. Cosa danno gli altri valori in A? Ce ne sono 255 da provare...

Per configurazioni diverse, modificate le varie linee LD A, 7 sostituendo altri valori da caricare in A: per esempio LD A, 8 oppure LD A, 32 o ancora LD A, 31 (tutti valori decimali). Al posto di 07 in effetti si può sostituire qualsiasi numero esadecimale a due cifre, e si possono usare valori diversi nei tre casi.
Per "pulire" inizialmente lo schermo, STOP (usando l'opzione "a"), poi fate CLS, e poi GO TO 300, per scegliere infine l'opzione "r".
Siete capaci di scoprire come funziona? e che cosa fa il programma?

Risposte

1. POKE 32000, 16
2. POKE 32000, 24
3. POKE 32000, 136
4. POKE 32000, 112
5. Modificare 5800 in 5801 (3A 00 7D 32 01 58)
6. Modificare 5800 in 5867 (3A 00 7D 32 67 58).

CAPITOLO QUINDICI

La memoria dello schermo

La cosa più importante da ricordare è che la memoria dello schermo principia all'indirizzo 4000 esadecimale, ed è lunga 1800(esadecimale) byte; e che, come abbiamo visto, si divide in tre blocchi, che vanno da 4000 a 47FF, 4800-4FFF e 5000-57FF rispettivamente.

Il modo migliore per prenderci un po' l'abitudine è di fare qualche esperimento, modificando la memoria dello schermo in vari punti, e notando i relativi risultati.

Ad esempio, supponiamo che collochiate lo stesso valore a ciascuno dei 6144 indirizzi. Cosa succede?

Per stare sul definito, poniamo che il valore sia 174 (decimale), ovvero AE (esadecimale) o 10101110 (binario). Possiamo usare HL per puntare all'indirizzo nella memoria dello schermo, e BC come contatore di cicli. Così avremo:

```
        LD HL, d-file       21 00 40
        LD BC, 1800         01 00 18
loop:   LD (HL), 174 dec    36 AE
        INC HL              23
        DEC BC              0B
        CP B                B8
        JRNZ loop           20 F9
        CP C                B9
        JRNZ loop           20 F6
```

Quello che si ottiene è una serie di strisce verticali. Esse sono formate in corrispondenza alla configurazione di bit del valore 174 in binario (10101110): trovate infatti una riga verticale nera ampia un pixel, poi una riga bianca ampia 1, poi un'altra coppia nera e bianca larghe 1 ciascuna, e quindi una riga nera più larga, ampia 3 pixel e infine una riga bianca ampia 1: il tutto che si ripete per 32 volte, da sinistra verso destra, lungo l'intero schermo.
Questo consegue al caricamento della stessa configurazione di bit entro ciascuna riga dello schermo, ripetuta 32 volte, con l'effetto di un allineamento verticale che fornisce le diverse strisce.

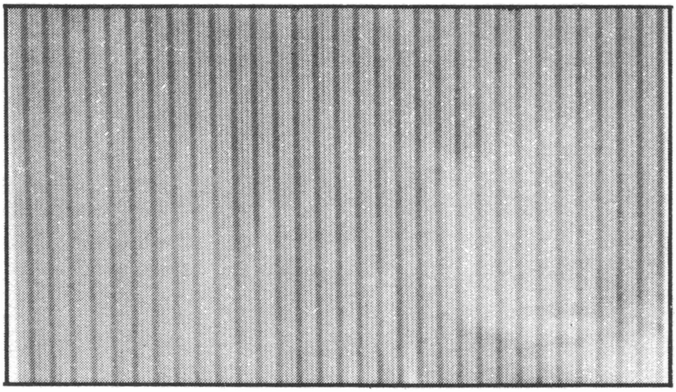


FIG. 15.1. Byte identici danno striature verticali replicate

Provate a modificare il 174 in altri numeri, e osservate che tipo di immagine ne ricavate. In particolare, provate con 1, 15 e 170, che in esadecimale corrispondono a 01, 0F e AA.

RIGHE ORIZZONTALI

Il programma successivo produce una serie di righe orizzontali, collocando il valore 255 nei byte relativi ad alcune righe dello schermo. Prima di usare l'opzione "r" è preferibile "pulire" lo schermo, nella maniera indicata alla fine del capitolo 14.

```
        LD A, 3            3E 03
        LD B, 0            06 00
        LD HL, d-file      21 00 40
loop:   LD (HL), 255 dec   36 FF
        INC HL             23
```

```
        DJNZ loop          10 FB
        DEC A              3D
        CP B               B8
        JRZ skip           28 08
        PUSH AF            F5
        LD A, 7            3E 07
        ADD A, H           84
        LD H, A            67
        POP AF             F1
        JR loop            18 EF
skip:   (RET instruction   C9)
```

SCHEMI DECORATIVI

Caricando byte differenti entro certe parti della memoria dello schermo è possibile creare immagini interessanti. Il programma che segue usa il registro C, che parte con 00 e si decrementa in FF, FE, FD, ... sino a 00 nuovamente (per tre volte in tutto), per definire il byte da caricare. Perciò, esaminando *accuratamente* lo schermo, potete scorgere il modo esatto in cui è ordinata la memoria dello schermo. In pratica, però, il quadro rende questa analisi confusa a meno che non sappiate già la risposta in anticipo...

```
        LD BC, 768 dec     01 00 18
        LD HL, d-file      21 00 40
loop:   LD (HL), C         71
        INC HL             23
        DEC BC             0B
        CP B               B8
        JRNZ loop          20 FA
        CP C               B9
        JRNZ loop          20 F7
```

Come variazione sul tema, provate a modificare la linea del *loop* per eseguire il caricamento dal registro B:

```
loop:  LD(HL), B        70
```

La configurazione cambia. Occorrono 256 iterazioni del ciclo per modi-

ficare il valore di B (dato che questo rappresenta il byte alto del contatore di cicli), corrispondenti a 8 linee in alta risoluzione sullo schermo. Il quadro rende chiaro come le prime otto righe abbiano valori diversi di B, cosicché i primi 256 byte *non* corrispondono alle righe 175-168 (come sarebbe se le righe ad alta risoluzione fossero manipolate in ordine dall'alto in basso); e il modo in cui la configurazione si ripete entro un blocco di 8 righe indica che i primi 256 byte in effetti contengono le righe 176, 167, 159 ecc., come è già stato spiegato in precedenza. L'intero quadro illustra bene la struttura a tre blocchi, come del resto in quello delle "righe orizzontali" del paragrafo precedente.

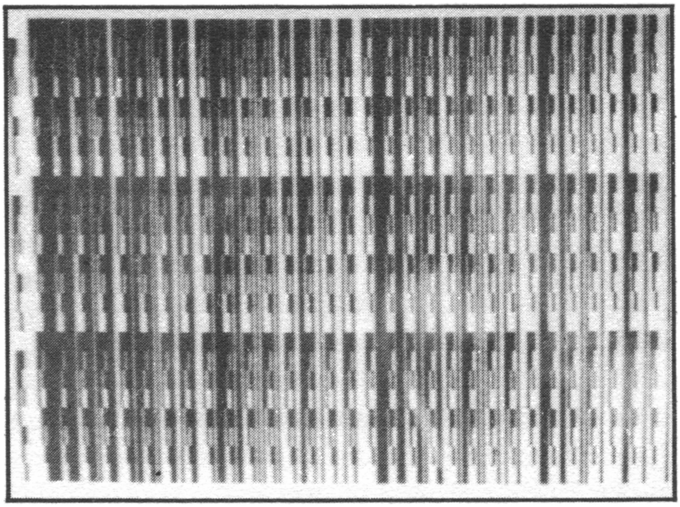


FIG. 15.2. Byte diversi danno strane configurazioni. Vedete i tre blocchi in cui si divide lo shermo? Siete capaci di ricavare i numeri binari che sono responsabili di questo quadro?

Programmi di questo tipo ci danno un po' di fiducia sul fatto che *siamo in grado* di fare cose interessanti tramite la memoria dello schermo, se proprio vogliamo; anche se non particolarmente *utili*, salvo che per creare questa fiducia. Perciò vediamo di accostarci a qualcosa che abbia scopi più determinati.

TRATTEGGIO

La routine che segue ha il vantaggio che per essa non ha grande importanza la struttura "fine" della memoria dello schermo. Essa scorre attraverso tale memoria e sostituisce ivi ogni byte zero (00000000) con il byte 10101010 (AA in esadecimale). L'effetto è di produrre un "tratteggio" delle zone vuote dello schermo con fini linee verticali.

```
       LD HL, d-file       21 00 40
       LD BC, 768 dec      01 00 18
       LD A, 0             3E 00
loop:  CP A, (HL)          BE
       JRNZ skip           20 02
       LD (HL), AA hex     36 AA
skip:  INC HL              23
       DEC BC              0B
       CP B                B8
       JRNZ loop           20 F6
       CP C                B9
       JRNZ loop           20 F3
```

L'istruzione LD, 0 in realtà non è strettamente necessaria: A contiene di per sé 0 a meno che non vi venga caricato un altro valore. Comunque serve a rendere leggermente più chiaro il funzionamento del programma. Provate comunque a dimostrare che non è necessaria eliminandola. (Il modo più semplice è di usare POKE 32006, 0, che converte tale linea in 00 00. Il codice 00 ha il significato di NOP – Nessuna OPerazione – e non serve a nulla, altro che a creare un breve ritardo. Il che lo rende ideale per sperimentare la cancellazione di certe istruzioni, perché tutti gli altri codici non richiedono alcuno spostamento entro la memoria).

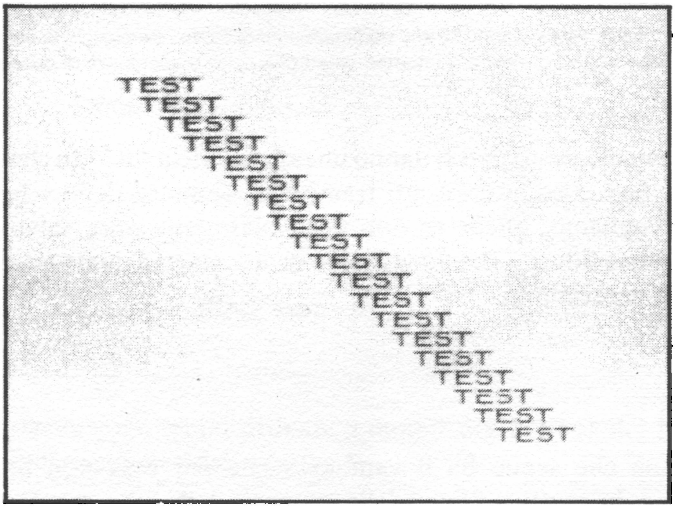


FIG. 15.3. Prima del "tratteggio"...

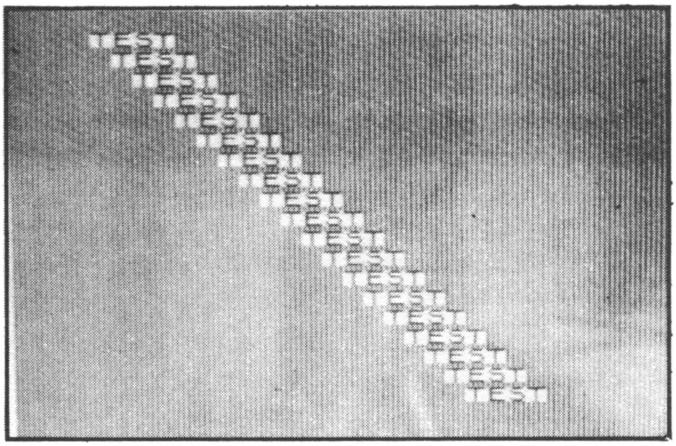
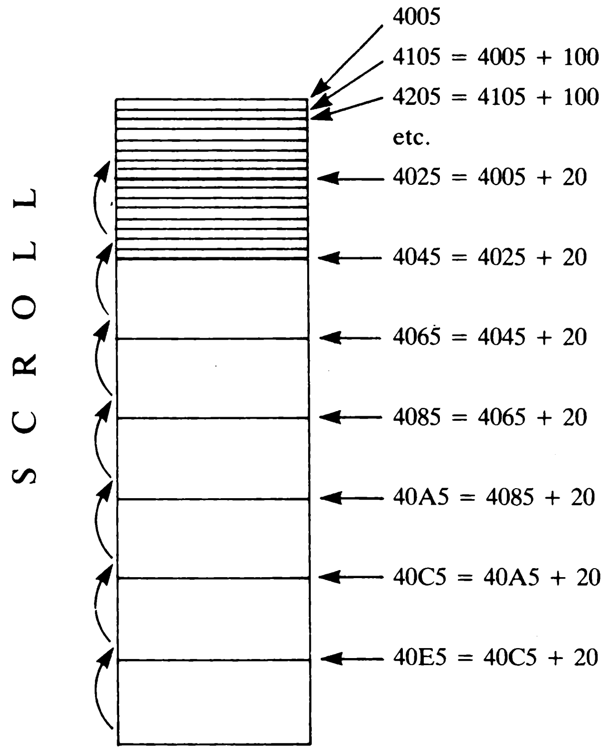


FIG. 15.4. ... e dopo

"SCROLLING" DI UNA SINGOLA COLONNA

Ora passiamo a qualcosa un po' più complicato: come "scrollare" una singola colonna dello schermo. Per semplificare un po' il problema, eseguiremo lo scrolling in uno solo dei tre blocchi: ovverossia, eseguiremo lo scrolling di una sezione di 8 linee di tale colonna. Per essere precisi, opereremo sulla colonna 5 del blocco 1.
La colonna 5 ha questo aspetto: si presenta in sezioni di 8 linee, ciascuna corrispondente a un carattere (e a una posizione PRINT AT), e ciascuna linea è memorizzata agli indirizzi indicati nella pagina seguente.





Per evitare ogni confusione di terminologia, chiamerò *riga* (come per il **PRINT AT**) ogni quadratino di 8 linee, e ogni singola linea in alta risoluzione, appunto, *linea*. Allora, la linea in testa alla riga 0 è contenuta nell'indirizzo 4005 (esadecimale) della memoria dello schermo. Se si somma 32 decimale (20 esadecimale) otteniamo l'indirizzo della linea di testa della riga 1; e così via sino alla riga 7.
Per avere l'indirizzo della seconda linea della riga 0 dobbiamo sommare 256 decimale (100 esadecimale) all'originario 4005, ottenendo

$$4005 + 0100 = 4105$$

Continuando a sommare 20 (esadecimale) otteniamo la seconda linea delle altre sette righe. Poi si passa analogamente alla terza, quarta, ... linea finendo all'ottava linea (di ciascuna riga).
Questa è la struttura della colonna (blocco 1). Per effettuare lo spostamento in su di 1 riga per ogni byte, dobbiamo collocarlo a un indirizzo che si ottiene dal suo sottraendo 32 (decimale). Scegliamo di perdere comunque la riga di testa, e di lasciare la settima riga vuota alla fine. Siamo in una situazione molto simile allo scrolling degli attributi di una

colonna di cui abbiamo parlato in precedenza, salvo che qui dobbiamo iterare 8 volte per muovere tutte e 8 le linee di un quadratino. (Una routine che servisse solo a spostare in alto l'ottavo superiore di ciascun carattere non sarebbe di molta utilità – specie se ricordiamo che esso è sempre una linea senza pixel colorati in INK!)
Dobbiamo quindi usare l'indicizzazione, con un offset di 0 o 32, come in precedenza.
Nella debole speranza di rendervi più chiaro il tutto, permettetemi prima di darvi una versione Basic della routine.

```
2000  LET ix = 256 * 64 + 5
2010  FOR l = 1 TO 8
2020  FOR a = 1 TO 7
2030  LET b = PEEK (ix + 32)
2040  POKE ix, b
2050  LET ix = ix + 32
2060  NEXT a
2070  POKE ix, 0
2080  LET ix = ix + 32
2090  NEXT l
```

Tanto per facilitare, ho usato nomi di variabili in minuscolo, come ix, corrispondenti ai nomi in maiuscolo dei relativi registri, come IX. Il ciclo FOR...NEXT basato su a sposta in su tutte le prime linee; la linea 2070 inserisce il carattere vuoto nella settima riga; e il ciclo FOR...NEXT basato su 1 ripete il procedimento per la seconda, terza, ..., ottava linea di ogni carattere.
Per visualizzare qualcosa, date LIST 2000 seguito da GO TO 2000.
Se fate la prova, vedrete che funziona – però è piuttosto lento: si vedono chiaramente i caratteri che "ondeggiano" verso l'alto.
Ad ogni modo siamo sulla strada giusta. (Per inciso, quello che abbiamo fatto è usare un utile strumento per il *debugging*, cioè il controllo dei difetti di un programma: prima scrivetene una versione Basic, per verificare se l'idea funziona; se necessario, correggetene i difetti; e solo dopo passate a tradurlo in versione in linguaggio macchina).
La trasformazione del Basic in linguaggio macchina porta a:

```
         LD DE, 0020      11 20 00
         LD IX, 4005      DD 21 05 40
         LD L, 08         2E 08
loop 2:  LD A, 07         3E 07
```

```
loop 1: LD B, (IX + 20)      DD 46 20
        LD (IX), B           DD 70 00
        ADD IX, DE           DD 19
        DEC A                3D
        CP D                 BA
        JRNZ loop 1          20 F4
        LD (IX), 00          DD 36 00 00
        ADD IX, DE           DD 19
        DEC L                2D
        PUSH AF              F5
        LD A, 0              3E 00
        CP L                 BD
        POP AF               F1
        JRNZ loop 2          20 E4
```

Per farne la prova, aggiungete al solito Basic:

```
1000 FOR i = 0 TO 21: FOR j = 0 TO 31:
  PRINT CHR$(65+i):NEXT j: NEXT i
```

Poi date GO TO 1000; e quindi fate partire la routine (o con GO TO 300 oppure con un RANDOMIZE USR 32000 in modo comandi). Il blocco superiore della colonna 5 si sposta effettivamente in su, ad una velocità rispettabile. Se iterate il codice macchina dal Basic ricaverete uno scrolling accettabilmente veloce:

```
3000 FOR k = 1 TO 8:RANDOMIZE USR 32000:NEXT k
```

Se la stessa iterazione è eseguita tramite linguaggio macchina, va così veloce che non riuscirete nemmeno a scorgere i caratteri che scompaiono.

"SCROLLING" MULTICOLONNA

Ora che siamo riusciti a far funzionare questo, è facile mettere assieme una routine che serve a "scrollare" un'intera sezione composta di più colonne, entro un dato blocco. Ci serve tuttavia un po' più di lavoro preparatorio.

Riservatevi quattro byte per dati in 7D00-7D03. Essi conterranno il numero della colonna iniziale, del blocco, il numero di colonne da scrollare, ed uno zero che funge solo da marcatore.

7D00	colonna	(per esempio 05 per colonna iniziale numero 5)
7D01	blocco	(40, 48 o 50 per ognuno dei tre blocchi)
7D02	larghezza	(per esempio 11(esadecimale) per 17(decimale) colonne)
7D03	00	

Il linguaggio macchina, che principia da 7D04 (e viene quindi chiamato mediante RANDOMIZE USR 32004) è il seguente: notate che la parte di mezzo è semplicemente una ripetizione di quanto abbiamo realizzato prima.

```
         LD BC, (7D02)        ED 4B 02 7D
         LD DE, 0020          11 20 00
         LD IX, (7D00)        DD 20 00 7D
loop 3:  PUSH IX              DD E5
         LD L, 08             2E 08
loop 2:  LD A, 07             3E 07
loop 1:  LD B, (IX + 20)      DD 46 20
         LD (IX), B           DD 70 00
         ADD IX, DE           DD 19
         DEC A                3D
         CP D                 BA
         JRNZ loop 1          20 F4
         LD (IX), 00          DD 36 00 00
         ADD IX, DE           DD 19
         DEC L                2D
         PUSH AF              F5
         LD A, 0              3E 00
         CP L                 BD
         POP AF               F1
         JRNZ loop 2          20 E4
         POP IX               DD E1
         DEC C                0D
         PUSH AF              F5
         LD A, 0              3E 00
         CP C                 B9
         POP AF               F1
         JRZ skip             28 04
         INC IX               DD 23
         JR loop 3            18 D2
skip:    (RET                 C9)
```

Per poterlo usare, fate il POKE dei vari numeri nei byte per i dati (per esempio 5, 64, 17, 0 – ricordate che per i POKE si devono usare i valori decimali, non esadecimali!). Poi usate GO TO 1000, che crea qualcosa sullo schermo, per potere fare le prove dello scrolling. Poi al solito o scegliete l'opzione "r" dopo GO TO 300, oppure date direttamente RANDOMIZE USR 32004. Ed ecco che tutto si sposta in alto di una riga!

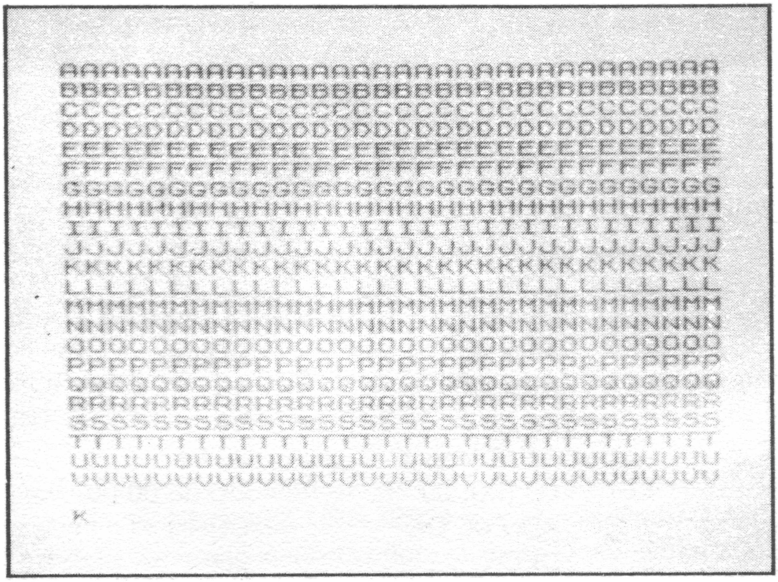


FIG. 15.5 Prima di eseguire lo "scroll" di una sezione dello schermo...

Provate a iterare l'effetto del Basic: il risultato è veramente efficace. Potreste impiegare una routine di questo tipo, adattata per lavorare sul blocco 2, per esempio, per realizzare una simpatica versione di una "Slot Machine"...
Un'osservazione finale. Quando ho provato per la prima volta questa routine, ho fatto un piccolo sbaglio nell'ultima parte del listato: ho messo il POP AF *dopo* l'istruzione JRZ skip. Il risultato è stato che (a) lo scrolling avveniva in maniera del tutto regolare, però (b) compariva il messaggio di errore C: Nonsense in Basic 0:1.

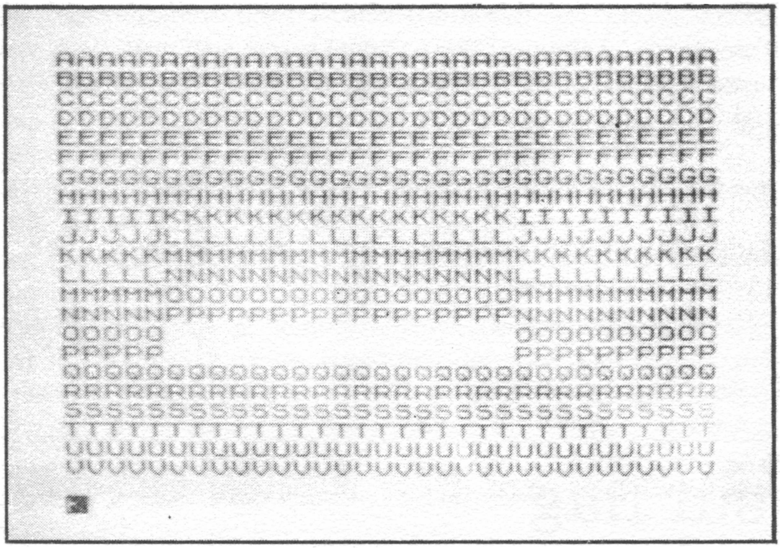


FIG. 15.6. ... ed il risultato, dopo due "scroll"

La stessa cosa potrebbe capitare anche a voi. È un'indicazione che qualcosa è andata storta nello stack. Infatti, con le istruzioni messe in quell'ordine, l'ultimo JRZ skip salta il POP AF; quando lo Spectrum tenta di rientrare al Basic, l'indirizzo di ritorno rinvenuto sullo stack (indirizzo formato da 2 byte) è quel resto del registro AF che ancora si trova lì, con il risultato di confondere completamente il sistema. È anche troppo facile che capiti di uscire dal linguaggio macchina con parti dello stack che non hanno subito il corretto POP: ma si tratta sempre di uno sbaglio.

CAPITOLO SEDICI

Ancora sui flag

Fino ad ora ho cercato di procedere tralasciando gli aspetti tecnici legati al registro F: però va precisato che ogni volta che abbiamo avuto a che fare con i salti condizionati abbiamo in effetti fatto ricorso ai flag. A questo punto merita darci un'occhiata più da vicino. Non desidero certo entrare nei minuti e fastidiosi dettagli: l'esatto funzionamento del registro F è una delle caratteristiche più complicate dello Z 80. Però non è sbagliato fissare qualche altro concetto.
Nel registro F c'è posto per otto bit: ma solo sei di questi vengono usati in pratica. Essi sono:

C	Flag del riporto (Carry)
Z	Flag di risultato Zero
S	Flag del Segno
P/V	Flag di Parità/oVerflow
H	Flag del semi-riporto (Half-carry)
N	Flag della sottrazione

La loro posizione nel registro F è questa:

S	Z	X	H	X	P/V	N	C

(X è un bit non utilizzato).
Il flag del Carry è modificato principalmente per effetto delle istruzioni di somma, sottrazione, rotazione e spostamento.

Il flag Zero viene modificato quasi in continuazione! "Grosso modo", se una qualche istruzione salvo LD, INC e DEC modifica il contenuto di A, allora il flag Z viene posto a 1 se A è zero, e posto a 0 negli altri casi. Anche BIT pone a 1 il flag Z se il bit verificato vale zero. CP "setta" o "resetta" questo flag a seconda del risultato del confronto.
Il flag del Segno conserva il bit del segno del risultato della più recente operazione eseguita: 1 per risultato negativo, 0 per positivo.
Il flag P/V è attivato in parte da operazioni aritmetiche, in parte da quelle logiche. Nel caso delle operazioni aritmetiche, viene posto a 1 se interviene un "overflow" in notazione con complemento a 2 (per esempio, se la somma di due numeri positivi "deborda" dall'estremo dell'accumulatore, dando apparentemente un risultato negativo). Nel caso delle operazioni logiche, esso viene posto a 0 se il byte posto in A ha un numero *pari* di bit eguali a 1; e, inversamente a 1 se il numero di bit eguali a 1 è *dispari*. Ecco il motivo del termine "parità". Per fare degli esempi:

Se A contiene 01101100: parità "pari": flag P/V posto a 0
Se A contiene 10010001: parità "dispari": flag P/V posto a 1

I flag H e N vengono impiegati solo per i calcoli cosiddetti BCD (*Binary Coded Decimal*, cioè decimali codificati in binario), ed è veramente poco probabile che dobbiate ricorrerci con lo Spectrum. Servono soprattutto per comandare unità periferiche esterne. La maniera in cui le varie operazioni modificano i più importanti flag C e Z è illustrata nell'appendice 5, a beneficio dei principianti.
Il flag Zero è stato utilizzato in diverse routine, e precisamente ogni volta che abbiamo usato JRZ, JRNZ o DJNZ, per esempio.
Di solito queste istruzioni di salto condizionato sono precedute da un'istruzione di *confronto*, come

```
CP B
```

che *posiziona i flag* nello stesso modo in cui farebbe l'istruzione SUB A, B; senza però in realtà modificare il contenuto di A, come farebbe quella. Se A e B contengono il medesimo byte, la sottrazione fornirebbe 0 come risultato, e il flag Zero viene posto a 1 ("settato"). Se A e B sono differenti, il flag Zero viene posto a 0 ("resettato").
Non abbiamo invece fatto largo uso finora del flag Carry. Esso risulta utile specie come contatore di cicli, quando non si sa con precisione se si raggiungerà esattamente la fine del ciclo, che forse verrà superata.
Per fare un esempio specifico, supponiamo che si voglia sapere se il numero contenuto nel registro HL è maggiore di quello nel registro BC.
L'istruzione

```
SBC HL, BC            ED 42
```

posizionerà a 1 il flag del Carry se BC è maggiore di HL; e lo porrà a 0 se BC risulta inferiore o eguale a HL. Se si fa seguire un salto condizionato

```
JR C indirizzo          38 xx xx
```

il salto avrà luogo se BC è maggiore di HL. Se invece segue

```
JR NC indirizzo          30 xx xx
```

il salto verrà eseguito se BC è eguale o minore a HL. (Per decidere sul rapporto di grandezza fra due numeri, questi sono sempre considerati in valore assoluto, cioè positivi: niente aritmetica in complemento a 2 qui!)

RINUMERAZIONE DELLE LINEE DI UN PROGRAMMA

Vedrò di illustrare alcuni di questi concetti sulla base di un rudimentale programma di rinumerazione dei numeri di linea di un programma Basic. Quello che fa questa routine è scorrere lungo l'area del programma Basic, e modificarne i numeri di linea per dare la sequenza regolare 10, 20, 30, ... con passo costante di 10 sino a raggiungere l'ultima istruzione. (Potete trovare routine più complete e variate in diversi altri posti; qui mi limito a darvi un esempio molto semplice a solo scopo illustrativo.)
Per fare questo, è bene ricordare come sono memorizzati i numeri di linea Basic. Anche questo lo potete trovare altrove, ma per rendervi le cose più facili ve lo rammenterò.
Un programma Basic viene collocato in memoria fra gli indirizzi che sono indicati dalle variabili di sistema PROG e VARS. Se non ci sono i Microdrive, PROG vale 23755. Ogni linea di istruzioni ha questo formato:

NA	NB	LB	LA	(Istruzione)	13

(dove 13 è il codice ASCII per l'ENTER finale, mentre "Istruzione" sta per la successione di codici ASCII o Sinclair per l'istruzione, semplice o multipla che sia). In essa, NA ed NB stanno per il byte alto ed il byte basso del numero di linea; mentre LB ed LA sono il byte basso ed alto della lunghezza complessiva (numero di caratteri) dell'istruzione (incluso l'ENTER finale, ma esclusi i primi 4 byte, NA, NB, LB e LA). Notate

come venga per primo il byte alto NA del numero di linea: *non* si tratta di un errore di stampa.
Per fare un esempio, il programma puramente sperimentale

```
1 REM 12345678901
5 PRINT a
```

si trova memorizzato come segue:

Indirizzo	Contenuto	Significato
23755	0	Byte alto del primo numero di linea
23756	1	Byte basso del primo numero di linea
23757	13	Byte basso della lunghezza della linea
23758	0	Byte alto della lunghezza della linea
23759	234	Codice di REM
23760	49	Codice per 1
23761	50	Codice per 2
23762	51	Codice per 3
23763	52	Codice per 4
23764	53	Codice per 5
23765	54	Codice per 6
23766	55	Codice per 7
23767	56	Codice per 8
23768	57	Codice per 9
23769	58	Codice per 0
23770	49	Codice per 1
23771	13	Codice di ENTER
23772	5	Byte alto del secondo numero di linea
23773	5	Byte basso del secondo numero di linea
23774	3	Byte basso della lunghezza della linea
23775	0	Byte alto della lunghezza della linea
23776	245	Codice di PRINT
23777	97	Codice per a
23778	13	Codice di ENTER
23779		Inizio area delle variabili

Potete verificare il tutto scrivendo un semplice programmino che in successione effettui il PEEK di tutte le locazioni di memoria comprese fra 23755 e 23779, e ne stampi ordinatamente il contenuto. Accertato questo, la maniera più ovvia per eseguire la rinumerazione delle linee è di andare alla ricerca, lungo l'area del programma, del codice 13 (per ENTER): i due byte successivi saranno il numero di linea: e basterà modificarli appropriatamente.
Ottimo... ma non funziona: il motivo è che il fatidico 13 può essere presente in diversi punti senza quel significato di ENTER: vedetene un esempio nella locazione 23757, dove indica la lunghezza di un'istruzione (naturalmente, l'ho fatto apposta per chiarirvi il problema). *C'è modo* di aggirare la difficoltà, ma è un po' complicato. (Nel buon vecchio ZX 81 il codice di NEWLINE era 118: in media, le istruzioni singole dello ZX

81 non superavano la lunghezza di 118, e il problema quasi non sorgeva.)
Inoltre, c'è un modo migliore per affrontare il problema. I byte che indicano la lunghezza della linea vi *segnalano* di quanti posti dovete spostarvi per trovare il prossimo numero di linea. Perché quindi cercare un equivoco 13 per ENTER?
E così arriviamo al programmino in linguaggio macchina che ora vi descriverò.

IL PROGRAMMA

L'idea consiste nel partire dai primissimi due byte dell'area del programma, che rappresentano il numero della prima istruzione; rinumerare questa; utilizzare il valore indicato dai due byte successivi per saltare direttamente al prossimo numero di linea; e iterare il tutto sino a che si raggiunge l'area delle variabili, ovvero l'indirizzo segnato da VARS.
Per prima cosa ci conviene memorizzare gli indirizzi opportuni. Useremo BC per contenere il valore di VARS (dove dobbiamo arrestarci), HL per contenere PROG (da dove partiremo), e DE per contenere ogni volta il nuovo numero di linea. Procediamo ad inizializzare tali registri:

```
LD BC,(VARS)          ED 4B 4B 5C
LD HL,(PROG)          2A 53 5C
LD DE, 10 decimale    11 0A 00
```

Per i valori da attribuire agli indirizzi delle variabili di sistema PROG e VARS si veda l'appendice 3.
Poi, dobbiamo controllare se il valore del registro HL (che useremo come puntatore che scorre lungo l'area del programma, dato che HL si presta ottimamente all'indirizzamento indiretto) è superiore a quello del registro BC. Se è così, ci fermiamo. *Qui* interviene il flag del Carry:

```
test:  PUSH HL        E5
       SBC HL, BC     ED 42
       POP HL         E1
       JRNC fine      30 15
```

(Quest'ultimo 15 in realtà non si potrebbe definire prima di aver concluso la scrittura dell'intero programma, ma vi assicuro che alla fine risulta il valore giusto.)
Quindi, l'effettivo lavoro di rinumerazione:

```
rinum:  LD(HL), D        72
        INC HL           23
        LD(HL), E        73
```

Dopo di che dobbiamo leggere i due byte successivi nell'area programma per stabilire la lunghezza della linea in cui stiamo. E dobbiamo salvare il risultato in qualche posto. Il registro HL ci serve per i calcoli; pertanto sembrerebbe idoneo il registro BC. Purtroppo anche questo viene già utilizzato; però possiamo aggirare l'ostacolo salvandone il valore corrente sullo stack, per poi recuperarlo in seguito:

```
PUSH BC      C5
INC HL       23
LD C,(HL)    4E
INC HL       23
LD B,(HL)    46
```

E ora spostiamo HL a indicare il prossimo numero di linea:

```
ADD HL, BC   09
INC HL       23
```

e recuperiamo il precedente valore di BC

```
POP BC       C1
```

A questo punto vogliamo che DE indichi il corretto nuovo numero di linea, sommando 10. Bisogna ricorrere ancora a un certo rimescolamento di registri con l'ausilio dello stack:

```
PUSH HL                 E5
LD HL, 10 decimale      21 0A 00
ADD HL, DE              19
LD D, H                 54
LD E, L                 5D
POP HL                  E1
```

Ora ricicliamo al punto test, per ripetere il procedimento sino a che HL non raggiunge VARS:

```
JR test          18 E5
```

Infine, il solito RET finale, che corrisponde al punto che in precedenza abbiamo indicato come fine.

```
   5 CLEAR 31999
   6 POKE 23609,50
  10 PRINT "base address ";
  15 DIM h$(2)
  20 INPUT b: PRINT b
  27 PRINT "No. of data bytes ";
  40 INPUT d: PRINT d
  50 FOR i=0 TO d-1
  55 FOR i=0 TO d-1
  60 POKE b+i,0
  70 NEXT i
  80 LET a=b+d
  87 PRINT "CODE:"
 100 INPUT c$
 106 IF c$="s" THEN GO TO 200
 110 IF c$="S" THEN GO TO 200
 120 PRINT c$+" ";
 128 LET hs=CODE c$(1)-48-39*(c$
(1)>"9")
 140 LET hj=CODE c$(2)-48-39*(c$
(2)>"9")
 150 POKE a,16*hs+hj
BEFORE       L
```

FIG. 16.1. Prima della rinumerazione delle linee...

```
  10 CLEAR 31999
  20 POKE 23609,50
  30 PRINT "base address ";
  40 DIM h$(2)
  50 INPUT b: PRINT b
  60 PRINT "No. of data bytes ";
  70 INPUT d: PRINT d
  80 FOR i=0 TO d-1
  90 FOR i=0 TO d-1
 100 POKE b+i,0
 110 NEXT i
 120 LET a=b+d
 130 PRINT "CODE:"
 140 INPUT c$
 150 IF c$="s" THEN GO TO 200
 160 IF c$="S" THEN GO TO 200
 170 PRINT c$+" ";
 180 LET hs=CODE c$(1)-48-39*(c$
(1)>"9")
 190 LET hj=CODE c$(2)-48-39*(c$
(2)>"9")
 200 POKE a,16*hs+hj
AFTER        L
```

FIG. 16.2. ... e dopo. Notate come i numeri dei GO TO sono immutati

Procedete al caricamento di tutta la routine, nell'ordine indicato (con il RET finale automatico). Ogni programma Basic in memoria verrà ora rinumerato se date il comando

RANDOMIZE USR 32000

È ovvio che solo i numeri di linea verranno modificati: eventuali numeri posti dopo GO TO e GOSUB non vengono infatti toccati, e i riferimenti che essi danno saranno sbagliati. Ma quel che contava era di chiarirvi l'idea di principio del metodo.

CAPITOLO DICIASSETTE

Ricerca e trasferimento di blocchi

Alcune fra le routine che vi ho presentato sin qûi non sono scritte nel modo più efficiente. L'idea era rendere le cose più semplici possibile all'inizio, e non sento di dovervi delle scuse per questo. Il linguaggio macchina non è semplice da affrontare, e darvi tutte assieme le sue caratteristiche sin dal principio avrebbe confuso le idee. D'altro canto, se avete avuto modo di dare uno sguardo ad altri libri sul linguaggio macchina dello Z 80, o ai listati in linguaggio macchina comparsi su qualche rivista, vi sarete forse meravigliati del perché io abbia in certi casi risolto certi problemi in maniera alquanto banale. Questo, e il successivo capitolo serviranno a riequilibrare un po' le cose.

RICERCA ENTRO I BLOCCHI

Innanzitutto, ci sono alcune istruzioni molto potenti che eseguono la ricerca entro blocchi completi di memoria. Come loro esempio prenderò CPDR che sta per "confronta, decrementa e ripeti".
Se scriviamo un piccolo programma come questo:

```
LD BC, 0100        0100 01
LD HL, 5000        21 00 50
LD A, 05           3E 05
CPDR               ED B9
seguito: ...
```

vediamo quel che succede.
Quando si incontra l'istruzione CPDR, si fa il confronto fra il valore di A e quello della locazione a cui punta il registro HL. Se sono eguali, il controllo passa alla parte seguito; se no, sia BC che HL vengono decrementati (di 1), e l'operazione di confronto viene ripetuta sino a che si trova la corrispondenza, oppure sino a quando BC raggiunge lo 0. In altre parole, queste quattro istruzioni significano: "Trovare la prima occorrenza del valore 05 nell'area che inizia dall'indirizzo 5000(esadecimale) sino all'indirizzo 4F00, e uscire con HL che punta a tale indirizzo. Se non si trova, BC vale zero".
Pertanto, una simile ricerca, data come primo esempio dell'uso dei salti e che utilizzava un piccolo ciclo di confronti, si poteva realizzare in modo molto più semplice così. Ovvio che allora però non sarebbe servita a dare un esempio di uso dei salti!
Esiste un'istruzione corrispondente, CPIR, che *incrementa* a ogni passo il valore di HL, ma per il resto funziona allo stesso modo.
Essa può servire a compiere ricerche entro un blocco partendo dalla fine.

TRASFERIMENTO DI BLOCCHI

Le istruzioni per il *trasferimento di blocchi*, LDDR e LDIR, servono a spostare interi blocchi di memoria. Ad esempio, per usare LDIR:
1. si carica l'indirizzo del primo byte da trasferire in HL;
2. si carica l'indirizzo del primo byte di destinazione in DE;
3. si carica il numero totale di byte da trasferire in BC.

Allora LDIR provvede al trasferimento del primo byte: poi incrementa HL e DE e decrementa BC; e continua da capo sino a che BC non raggiunge zero.
Un punto importante da notare in proposito è che nell'ultimo passo si ha l'incremento di HL e DE, ma *non* il trasferimento del byte successivo. Quindi, al termine, HL punta all'indirizzo immediatamente successivo all'area di origine, mentre DE punta al termine della regione di destinazione. (Vedere le successive routine per lo scrolling laterale).
LDDR opera in modo del tutto simile, con la differenza che HL e DE vengono decrementati di 1 ad ogni passo (mentre BC seguita ad essere decrementato come per LDIR sino a raggiungere lo zero – BC funge infatti esclusivamente da contatore).

PREDISPOSIZIONE DEGLI ATTRIBUTI

Uno dei modi per utilizzare il trasferimento di blocchi è predefinire una

"finta" memoria degli attributi in un posto idoneo della RAM, e poi trasferirla nella vera collocazione della memoria degli attributi, modificando così pressocché istantaneamente tutti gli attributi dell'immagine sullo schermo in una maniera predeterminata ad arte. Per fare questo, occorre riservare 704 byte per contenere i "nuovi" attributi. Ci serve allo scopo una piccola modifica al programma di caricamento. Modificate la linea 10 in modo che ora sia

```
10  CLEAR 31599
```

lasciando così un ampio spazio libero. Ora date RUN, e segnalate che ci saranno 704 byte per i dati. Il codice macchina da caricare è:

```
LD HL, 31600 dec     21 70 7B
LD DE, a-file        11 00 58
LD BC, 704 dec       01 C0 02
LDIR                 ED B0
```

Il valore 31600 (7B70 in esadecimale) corrisponde all'inizio della nuova area per i dati. La routine va poi caricata a partire da 32304.
Ora dovete creare la "finta" memoria degli attributi. Per esempio in questo modo:

```
5000  FOR w = 31600 TO 32303
5010  IF w < 31900 THEN POKE w, 48
5020  IF w > = 31900 THEN POKE w, 32
5030  NEXT w
```

Date GO TO 5000 per mandarlo in esecuzione (ed aspettate qualche secondo!); poi realizzate una qualche immagine sullo schermo (il modo più semplice è di dare LIST), e infine eseguite RANDOMIZE USR 32304. Vedrete diverse zone colorate in verde ed in giallo, a seconda degli attributi che sono stati inseriti dai POKE del programma che comincia alla linea 5000. Certamente saprete escogitare altri e più spettacolari modi di predeterminare la memoria degli attributi via Basic (a strisce colorate? a scacchi?).

IL PUNTO DI PARTENZA

A questo punto, dovreste essere capaci di seguire il funzionamento dei programmini introduttivi che vi ho fornito nel primo capitolo. In man-

canza del programma di caricamento per il linguaggio macchina, in essi il linguaggio macchina era fornito sotto forma di DATA, scritti direttamente in decimale e non in esadecimale per facilitarne l'introduzione. Se convertite i vari numeri decimali in esadecimale e poi ne verificate il significato (una tabella degli opcode in ordine numerico, sia decimale che esadecimale, è fornita dal Manuale Sinclair con i relativi significati), vi accorgerete che tutti includono trasferimenti di blocchi entro la memoria degli attributi o dello schermo. I byte trasferiti vengono prelevati dalla ROM, di conseguenza il più delle volte hanno una configurazione piuttosto casuale. Alcune sezioni della ROM, però (sapete individuare quali?), hanno una struttura più ordinata, ed è da lì che derivano le immagini più strutturate.

"SCROLLING" LATERALE DEGLI ATTRIBUTI

Riportate la linea 10 del programma Basic di caricamento del linguaggio macchina a CLEAR 31999, perché adesso non ci serve più tutto quello spazio ausiliario per i dati.
La predeterminazione dei valori degli attributi è cosa semplice impiegando LDIR. Eccovi un metodo un po' più spettacolare: effettua lo scrolling di un posto verso sinistra di una riga degli attributi, riportando quello della casella all'estremo sinistro all'estremità opposta, come se lo schermo si riavvolgesse da destra come su una superficie cilindrica. Per semplicità, mi limiterò alla riga 0.

```
        LD HL, a-file     21 00 58
loop:   LD D, H           54
        LD E, L           5D
        INC HL            23
        LD BC, 31 dec     01 1F 00
        LD A, (DE)        1A
        LDIR              ED B0
        LD (DE), A        12
```

Come al solito, creiamo qualcosa sullo schermo per vedere l'effetto:

```
6000 FOR s = 0 TO 31:PAPER s/6:PRINT AT 0,6;" ";:NEXT s
```

e poi date il solito RANDOMIZE USR 32000: osservate come avviene lo scrolling della riga. Per un effetto più appariscente, create un ciclo iterativo in Basic

```
FOR t = 1 TO 500:RANDOMIZE USR 32000:NEXT t
```

e osservate come scorre veloce!
Se poi iterate quanto sopra 22 volte riferendovi alle diverse righe, con opportuni valori di partenza, potete far scorrere gli attributi dell'intero schermo. Ora affronterò un problema molto simile: effettuare lo scrolling laterale di una linea della memoria dello *schermo*; il che richiede comunque dei cicli iterativi.

"SCROLLING" LATERALE DELLO SCHERMO

Eccovi direttamente il codice macchina atto allo scopo:

```
        LD A, 8 dec.     3E 08      (contatore di ciclo)
        LD DE, start     11 00 40   (inizio linea)
set:    LD H, D          62
        LD L, E          6B
        INC HL           23
        PUSH DE          D5         (salva inizio linea)
        LD BC, 31 dec.   01 1F 00
        PUSH AF          F5         (salva contatore)
        LD A,(DE)        1A         (salva il primo byte)
        LDIR             ED B0      (scrolling a sinistra)
        LD(DE), A        12         (ricarica il primo byte)
        POP AF           F1         (recupera contatore cicli)
        DEC A            3D         (decrementa contatore)
        CP B             B8         (verifica se finito)
        JRZ skip         28 04
        POP DE           D1
        INC D            14         (linea seguente)
        JR set           18 EB
skip:   POP DE           D1
```

Se inserite questo in un altro ciclo (o più cicli, per i diversi blocchi e per 8 linee in ogni blocco), avrete una routine capace di eseguire lo scrolling laterale dell'intero schermo. Ovviamente sono possibili diverse varianti. Studiate attentamente questa routine, e vedete di trovare le modifiche più cpportune da provare.

CAPITOLO DICIOTTO

Alcune cose di cui finora non vi ho parlato

INVERSIONE DI BIT

Esistono alcune istruzioni per l'inversione dei bit che vi possono essere utili. La prima è

CCF 3F

che commuta il valore del flag del Carry (da 0 a 1, o da 1 a 0). Per posizionare invece il flag del Carry a 1 usate

SCF 37

e per rimetterlo a 0, usate le due in serie: prima SCF, poi CCF. Per "complementare" l'intero registro A (ossia per invertire ogni suo bit nell'opposto) si usa

CPL 2F

SIGLE MNEMONICHE

L'altra cosa che devo dirvi è che altri può usare sigle mnemoniche per i vari opcode diverse da quelle che vi ho descritto. Ad esempio, in certe parti potrebbe trovare, invece di LD A,(nn), un semplice LD (nn). Ciò

viene dal fatto che il registro A è l'unico registro che può venire caricato in modo diretto, e pertanto non è strettamente necessario specificarlo. Comunque, trovo che specificarlo esplicitamente ogni volta torni utile alla memoria.

REGISTRI ALTERNATIVI

Vi ho detto inizialmente che esiste una serie completa di registri alternativi, e poi li ho tranquillamente lasciati a dormire. In genere, potete arrangiarvi quasi sempre senza doverli usare, e comunque non sono particolarmente utili. In essi non potete eseguire dei calcoli aritmetici. Il loro impiego principale è per salvare temporaneamente i contenuti del gruppo principale di registri mentre si sta eseguendo qualche routine che modifica i contenuti di alcuni di essi in maniera non desiderabile. Il metodo consiste nello scambio fra il set principale e quello alternativo sia prima che dopo la routine in questione:

```
EX AF, AF'    08          scambia fra di loro AF e AF'
EXX           D9          scambia BC, DE, HL con BC', DE', HL'
CALL...       CD xx xx    chiamata della routine
EX AF, AF'    08          ristabilisce i registri
EXX           D9             originari
```

Naturalmente, potreste ottenere un risultato simile con il PUSH dei registri di cui si vuole salvare il contenuto, ponendoli entro lo stack, prima di effettuare il CALL, e richiamandoli poi ordinatamente con altrettanti POP.
Attenzione: non usate *mai* il registro IY: ne ha bisogno lo Spectrum!

LE ROUTINE DELLA ROM

Qua e là mi sono anche reso colpevole di avere reinventato l'ombrello. Infatti, l'interprete Basic posto nella ROM deve spesso ricorrere alla chiamata di routine simili a quelle che abbiamo messo a punto. E allora perché, più semplicemente, non effettuare la loro chiamata, invece di scriverne di proprie? In genere, la risposta sarebbe che è vero, sarebbe stato più ragionevole fare così, perché ci si risparmia un sacco di lavoro e, cosa forse più importante, parecchio spazio di memoria. Tuttavia – almeno per quanto concerne *questo* libro – il mio scopo era spiegarvi il linguaggio macchina dello Z 80, evitando, per quanto possibile, il ricorso alle speciali caratteristiche del sistema dello Spectrum.
Se tutti gli esempi fossero consistiti semplicemente in una serie di

chiamate a certi indirizzi della ROM, non avreste davvero imparato molto!

SALVATAGGIO DI PROGRAMMI BASIC+LINGUAGGIO MACCHINA

Facciamo l'ipotesi che abbiate preparato un programma Basic che utilizza una routine in linguaggio macchina collocata nell'area sopra la RAMTOP (diciamo, a partire dal nostro solito indirizzo 32000). Come potete salvare efficientemente entrambi i programmi?
Una delle maniere è di usare, di seguito

```
SAVE"programma"
SAVE"lm" CODE 32000,600
```

(dove 600 è la lunghezza dell'area del linguaggio macchina: potete usare in maniera più efficiente la vera lunghezza del codice in linguaggio macchina, ma sta a voi scegliere). Per il caricamento, dovete corrispondentemente fare prima

```
LOAD"programma"
```

e quando questo è stato caricato seguitare con

```
LOAD"lm" CODE
```

E non dimenticate prima di tutto il CLEAR 31999.
Ma ancora meglio è scrivere una piccola routine in Basic in calce al programma che faccia tutto questo per voi. Perciò, se avete un numero di linea libero verso la fine, diciamo 9000, scrivete

```
9000 LOAD"lm"CODE : GO TO 1
```
(o altro numero di linea da dove deve partire il programma)

Poi eseguite in modo diretto

```
SAVE"programma" LINE 9000
SAVE"lm"CODE 32000,600
```

Allora, quando, come di solito, date

```
LOAD"programma"
```

viene caricato il programma Basic, che parte automaticamente dalla

linea 9000, carica il linguaggio macchina, e poi prosegue regolarmente. Potete anche inserire un'altra linea Basic che esegua da programma il salvataggio secondo le modalità indicate, se volete. Provate.

ROUTINE MULTIPLE

Potete senz'altro "stipare" diverse routine in linguaggio macchina l'una dopo l'altra, purché prendiate sempre la precauzione di porre l'istruzione RET come ultima di ciascuna di esse: ognuna può distintamente venire poi chiamata col solito RANDOMIZE USR (seguito dall'indirizzo iniziale della routine pertinente). Notate che la serie di routine così disposte in un unico blocco si può salvare in una sola volta: *non* è necessario salvarle una per una.

IMPIEGO EFFICACE DEL LINGUAGGIO MACCHINA

Qui desidero riannodare le fila di due argomenti che ho lasciato pendenti verso l'inizio. Ho detto che vi sono altre ragioni, oltre a quella del tempo richiesto dall'interprete Basic per interpretare ogni singola istruzione al momento dell'esecuzione, perché il linguaggio macchina risulti più veloce del Basic. Chiarirò meglio facendo un esempio:

Basic		Linguaggio macchina
10 FOR i=20 TO 1 STEP-1		LD B, 14
......	loop:	.....
50 NEXT i		DJNZ loop

In entrambi i casi, ogni volta che viene eseguito il ciclo, una variabile viene decrementata di 1. Ma questo procedimento risulta molto più complicato in Basic che nel linguaggio macchina. Il motivo sta nel fatto che, poiché il Basic (Sinclair) deve spesso manipolare numeri non interi, esso finisce per agire così con tutti i tipi di numeri, e perciò sottrae in effetti il valore 1.00000000, il che per lui è come sottrarre 1.58716248. Il procedimento impiegato è piuttosto complicato e richiede un certo tempo. Il linguaggio macchina, d'altro canto, usa a tale scopo una singola istruzione appositamente concepita: il risultato è una velocità di esecuzione circa 100 volte maggiore!
L'altro argomento rimandato riguarda il fatto che il linguaggio macchina può talvolta occupare più spazio di memoria di un suo equivalente Basic.

Eccovi un esempio significativo:

Basic	Linguaggio macchina	Numero di byte
30 IF r = p AND p = q THEN LET p = w		
	LD HL, 5000	3
	LD A, (HL)	1
	LD HL, 5001	3
	SUB A, (HL)	1
	JRNZ nextbit	2
	LD HL, 5001	3
	LD A, (HL)	1
	LD HL, 5002	3
	SUB A, (HL)	1
	JRNZ nextbit	2
	LD HL, 5001	3
	LD A, (5003)	3
	LD (HL), A	1
	nextbit:	27 TOTALE

Il linguaggio macchina prevede che r, p, q e w siano rispettivamente contenuti agli indirizzi 5000, 5001, 5002 e 5003 (esadecimali). In pratica, le cose non sono così semplici, perché ciascun numero occupa cinque byte, e il SUB sarà in realtà una chiamata (CALL) a una routine per la sottrazione in virgola mobile. Ad ogni modo, l'effettivo codice in linguaggio macchina occuperebbe almeno altrettanti, e probabilmente di più, dei 27 byte illustrati. L'equivalente Basic occupa solo 18 byte, e cioè 1 per ciascuna parola chiave o simbolo (IF, =, w, AND ecc.); 4 per numero di linea e lunghezza, e 1 per il terminatore (ENTER). Naturalmente, quanto più complessa l'istruzione Basic, tanto maggiore risulterà la differenza di lunghezza per l'equivalente versione in linguaggio macchina.

ALTRI POSTI DOVE MEMORIZZARE IL LINGUAGGIO MACCHINA

Il principale svantaggio della collocazione di un codice in linguaggio macchina sopra la RAMTOP è, come abbiamo visto, che esso non può venire direttamente salvato assieme a un programma Basic da cui dipende.

Il vantaggio è che si può caricare dell'altro "sotto" ad esso. Esistono però delle alternative.
Un trucco a cui si ricorre spesso consiste nel memorizzare il linguaggio macchina in una istruzione REM, posta come prima linea del programma Basic.
Il primo byte subito dopo il REM ha solitamente per indirizzo 23760. Pertanto si può incominciare il programma Basic con una linea tipo

```
1 REM XXXXXXXXX...X
```

dove il numero delle X (o di un qualsiasi altro carattere) è sufficiente a coprire la lunghezza del codice in linguaggio macchina; dopo di che si inserisce in quelle posizioni il linguaggio macchina con dei POKE idonei. Il linguaggio macchina verrà richiamato con RANDOMIZE USR 23760 (oppure LET y = USR 23760 ecc.), ma, cosa importante, viene salvato assieme al Basic; inoltre non viene cancellato dal RUN (ma dal NEW sì, a differenza di quello posto oltre la RAMTOP).
Altro posto dove si può collocare del linguaggio macchina è in una stringa di caratteri, che può venir facilmente rintracciata (tramite la variabile di sistema VARS), purché la corrispondente variabile stringa sia la *prima* dichiarata in assoluto: cosa che si può fare per esempio dimensionando subito una stringa di lunghezza sufficiente per il linguaggio macchina:

```
1 DIM a$(79)          (per contenere 79 byte)
```

per poi inserire il linguaggio macchina in a$(1), a$(2) ecc. a mano a mano che viene creata la stringa. In questo caso, c'è lo svantaggio che RUN o CLEAR cancellano completamente il linguaggio macchina. In certi casi, inoltre, l'inizio del linguaggio macchina può "andare a spasso" nella memoria. Esiste infine la possibilità di memorizzare il linguaggio macchina entro dei DATA, da far caricare poi ad opera del Basic sopra la RAMTOP (proprio come abbiamo fatto nel capitolo 1), ma questo metodo comporta un grande spreco di memoria.

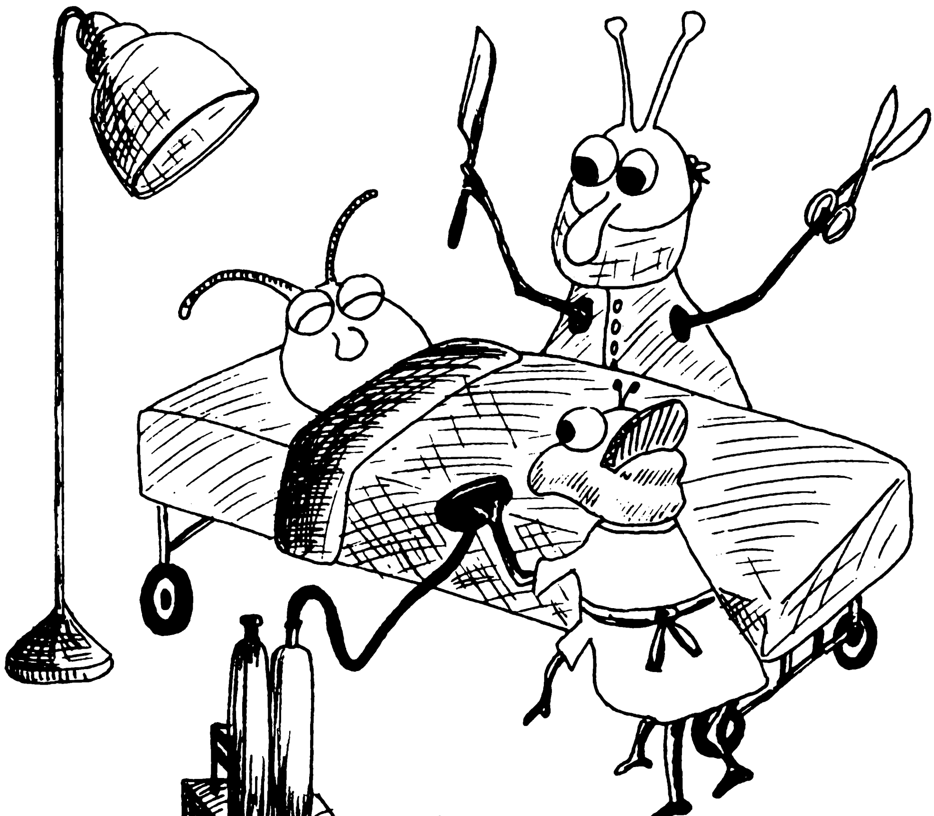
IDENTIFICAZIONE E CORREZIONE DEGLI ERRORI ("DEBUGGING")

Nel linguaggio macchina non sono previsti strumenti interni per effettuare la ricerca e la correzione degli eventuali errori di programmazione, ossia per effettuare quello che con pittoresco termine viene chiamato il "*debugging*" (si potrebbe tradurre "spulciamento"). Anche se HELPA potrà esservi di aiuto nelle modifiche al codice, esso non è stato espressamente concepito per fare del *debugging*. (E non lo sono neppu-

re diversi dei programmi falsamente pubblicizzati come utili per il *debugging* di routine in linguaggio macchina!). Il modo migliore di cavarsela a questo punto è eseguire "sulla carta", con foglio e matita, il programma, per controllarne con cura il funzionamento in base a un esempio pratico. Naturalmente, potete inserire apposite istruzioni di avvertimento dei vari punti raggiunti nel vostro linguaggio macchina, ma state attenti alle modifiche negli indirizzi assoluti e negli offset dei salti quando inserite o eliminate successivamente tali istruzioni.
Un metodo utile (anche se apparentemente banale) è quello di scrivere prima la routine in Basic, e procedere al *debugging* di *questa*: è preferibile usare solo istruzioni Basic che corrispondano al linguaggio macchina (il che si dice "emulare" il linguaggio macchina in Basic). Se il programma funziona, sarà lento: ma almeno potrete correggerlo con mezzi noti.
Per chi ha ambizioni, la cosa suggerisce un progetto da sviluppare. L'idea è di migliorare HELPA aggiungendo una routine PASSO-PASSO, che procede attraverso il programma mentre è in esecuzione passo dopo passo (istruzione dopo istruzione), mostrando i contenuti dei vari registri sullo schermo. Vi sarà necessario (a) scrivere una routine in linguaggio macchina che salvi con dei PUSH tutti i registri sullo stack, e poi ne visualizzi il contenuto in esadecimale sullo schermo; (b) aggiungere una routine che provveda a verificare se c'è un INPUT dalla tastiera; (c) inserire fra le diverse istruzioni successive della routine principale un CALL a questa routine (usate i marcatori ** di HELPA per segnalare dove); (d) prevedere il modo di tornare al Basic se ritenuto utile.

Innanzitutto - si trattava di Humpty-Dumpty*

Appendici



1 Conversione esadecimale/decimale

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
A	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
B	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
C	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
D	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
E	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
F	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

complemento a 2 (righe da 8 a F negative, fino a 7 positive)

ordinaria (righe da 0 a F)

2 Tabelle per riservare una zona di memoria

Per riservare spazio in memoria, in blocchi di 100 byte in fondo alla RAM, si usino i seguenti indirizzi. Ricordatevi di effettuare il CLEAR all'indirizzo inferiore di 1 a quello effettivamente richiesto

Numero di byte da riservare	16 K		48 K	
	Indirizzo decimale	Indirizzo esadecimale	Indirizzo decimale	Indirizzo esadecimale
100	32500	7EF4	65268	FEF4
200	32400	7E90	65168	FE90
300	32300	7E2C	65068	FE2C
400	32200	7DC8	64968	FDC8
500	32100	7D64	64868	FD64
600	32000	7D00	64768	FD00
700	31900	7C9C	64668	FC9C
800	31800	7C38	64568	FC38
900	31700	7BD4	64468	FBD4
1000	31600	7B70	64368	FB70
1100	31500	7B0C	64268	FB0C
1200	31400	7AA8	64168	FAA8

3 Indirizzi delle principali variabili di sistema

PUNTATORI A INDIRIZZI MOBILI

	Decimale	Esadecimale
CHANS	23631	5C4F
PROG	23635	5C53
VARS	23627	5C4B
E-LINE	23641	5C59
WORKSP	23649	5C61
STKBOT	23651	5C63
STKEND	23653	5C65
RAMTOP	23730	5CB2
UDG	23675	5C7B
P-RAMT	23732	5CB4
CHARS	23605	5C36
BORDCR	23624	5C48
DFSZ	23659	5C6B
COORDS	23677–8	5C7D–E
DFCC	23684	5C84
DFCC-S	23685	5C85
SCR-CT	23692	5C8C
ATTR-P	23693	5C8D

INDIRIZZI FISSI NELLA MEMORIA	Decimale	Esadecimale
INIZIO MEMORIA DELLO SCHERMO	16384	4000
INIZIO MEMORIA DEGLI ATTRIBUTI	22528	5800
VALORE DI RAMTOP PER 16 K RAM	32599	7F57
VALORE DI RAMTOP PER 48 K RAM	65367	FF57
INIZIO AREA UDG PER 16 K RAM	32600	7F58
INIZIO AREA UDG PER 48 K RAM	65368	FF58
VALORE DI P-RAMT PER 16 K RAM	32767	7FFF
VALORE DI P-RAMT PER 48 K RAM	65535	FFFF
INIZIO AREA BUFFER DELLA STAMPANTE	23296	5B00
INIZIO ZONA VARIABILI DI SISTEMA	23552	5C00
INIZIO ZONA MICRODRIVE MAPS	23734	5CB6
INIZIO ZONA SET DI CARATTERI	15360	3C00

4 Sommario delle istruzioni dello Z 80

Eccovi un'elenco delle sigle mnemoniche dei codici operativi (opcode), con una breve descrizione del loro effetto. Per i comandi per i quali nel testo è fornita una descrizione più ampia abbiamo dato un riferimento alla pagina. Abbiamo omesso gli effetti sui vari tipi di flag: per questi, consultare la "Zilog Reference Card" o alcuni dei testi indicati in bibliografia sotto il soggetto "Linguaggio Macchina"

ADC	Pag.	Somma, incluso il flag del Carry. Il risultato va in A o HL.
ADD	Pag.	Somma, senza tener conto del flag Carry. Il risultato va in A o HL.
AND	Pag.	AND logico fra bit di posto eguale: il risultato va in A.
BIT	Pag.	BIT b,r posiziona il flag Zero a seconda del valore del bit di posto b nel registro r. I bit sono nell'ordine 76543210 in ciascun byte.
CALL	Pag.	Chiamata di una subroutine. Esistono anche chiamate condizionate dal valore di un dato flag, specificato da una lettera addizionale: C (chiamata se il flag del Carry è ad 1); M (se il flag del segno vale 1 – "il risultato (di un CP) è negativo"); NC (se il flag Carry è a 0); NZ (se il flag Zero vale 0); P (se il flag del segno è 0- "il risultato è positivo"); PE (se il flag di Parità vale 1: però trascurate questo caso); PO (se il flag di Parità vale 0: trascurate anche questo); Z (se il flag Zero vale 1). Per i flag, vedi le pagg. , .
CCF	Pag.	Commuta (da 0 ad 1, oppure da 1 a 0) il flag del Carry.
CP	Pag.	Confronta: posiziona i flag come se venisse effettuata la sottrazione da A, ma senza modificare A.
CPD		Confronta e decrementa. Il confronto avviene tramite HL; poi sia HL che BC vengono decrementati.
CPDR	Pag.	Confronta, decrementa e ripeti: per la ricerca nei blocchi. Come CPD, ma iterando sino a che il risultato del confronto è zero, oppure BC raggiunge lo 0.
CPI		Come CPD, ma qui HL viene incrementato (BC sempre decrementato).
CPIR	Pag.	Come CPDR, ma con l'incremento di HL.
CPL	Pag.	Inversione di tutti i bit del registro A.
DAA		Aggiustamento decimale dell'accumulatore. Impie-

		gato nei calcoli tipo BCD (decimale codificato in binario): da trascurare.
DEC	Pag.	Decrementa: ossia riduce di 1 il valore del registro.
DI		Disabilita le interruzioni (*interrupt*). Trascurare.
DJNZ	Pag.	Decrementa e salta se non zero. Decrementa B e effettua un salto relativo sino a quando il flag dello Zero non passa a 1. Usato nei cicli come il FOR/NEXT in Basic.
EI		Abilita ("*enable*") le interruzioni. Trascurare.
EX	Pag.	Scambia i valori. In particolare, istruzioni di questo tipo che comprendono (SP) scambiano il contenuto dei registri HL, IX o IY con il valore che si trova in cima allo stack.
EXX	Pag.	Scambia tutti e tre i registri BC, DE e HL con i corrispondenti alternativi BC′, DE′ e HL′.
HALT		Attendi un'interruzione. A meno che non abbiate collegato unità esterne, e sappiate quel che fate, *non usate* questa istruzione, perché il programma si fermerà all'infinito.
IM		Modalità interrupt: trascurare.
IN		INPUT da una periferica: trascurare per il momento.
INC	Pag.	Incrementa: aumenta di 1 il valore del registro.
IND		
INDR		
INI		
INIR		Comandi di input analoghi ad LDD, LDDR, LDI, LDIR. Trascurateli.
JP	Pag.	Salta. Le diverse varianti con aggiunta di C, M, NC, NZ, P, PE, PO e Z sono salti condizionati in relazione alle varie condizioni di stato dei flag, come citate sotto CALL.
JR	Pag.	Salto relativo – seguito da un byte che indica l'entità (in numero di byte) dello spostamento (*offset*). Come varianti condizionate sono previste qui solo C, NC, Z, NZ.
LD	Pag.	Carica. Sono possibili tutti e cinque i vari modi di indirizzamento.
LDD		Da non confondere con LD D! Carica nella locazione puntata da DE il contenuto della locazione puntata da HL; poi decrementa BC, DE e HL.
LDDR	Pag.	Carica (*load*), decrementa e ripeti: per il trasferi-

		mento di blocchi. Esegue un LDD sino a quando BC diventa zero. Copia un blocco di memoria la cui lunghezza è posta in BC, trasferendo a ogni passo dalla locazione a cui punta HL a quella puntata da DE.
LDI		Come LDD ma qui HL e DE vengono incrementati, mentre BC seguita a essere decrementato.
LDIR	Pag.	Come LDDR, salvo che HL e DE vengono incrementati.
NEG		Negazione: cambia di segno il contenuto di A.
NOP		Nessuna OPerazione. Non fa nulla per la durata di un ciclo di temporizzazione – ossia, crea un breve ritardo. Utile per cancellare temporaneamente alcune istruzioni nel corso del *debugging*: può servire, e non dà alcun disturbo.
OR	Pag.	OR logico sui bit di egual posto. Il risultato va in A.
OTDR		
OTIR		
OUT		
OUTD		
OUTI		Varie istruzioni afferenti all'uscita di segnali. Trascuratele.
POP	Pag.	Trasferimento dalla cima dello stack nel registro indicato (e aggiornamento del puntatore allo stack).
PUSH	Pag.	Trasferisci il contenuto del registro indicato sulla cima dello stack (e aggiornamento dello SP).
RES	Pag.	Riposizionamento di un bit – vale a dire, porlo a 0.
RET	Pag.	Ritorno da una subroutine. Sono possibili istruzioni di rientro condizionato, corrispondenti alle varianti descritte sotto CALL. (Le condizioni prevalenti al momento del CALL non sono necessariamente le stesse quando si raggiunge il relativo RET!)
RETI		
RETN		Ritorno da una subroutine di trattamento degli interrupt. Trascurare.
RL		Ruotare a sinistra: simile ad uno spostamento "orizzontale" ("*shift*"), salvo che anche il flag Carry viene incluso nello spostamento, come se fosse il bit numero 8.
RLA		Rotazione a sinistra dell'accumulatore. Come RL A, ma con differenti effetti sui flag.

RLC		Da non confondere con RL C! Rotazione a sinistra, ma ponendo il valore del bit numero 7 nel flag del Carry e *anche* nel bit 0.
RLCA		Come RLC A, ma con le stesse differenze rispetto ai flag che per RLA.
RLD		Non è quello che credete: significa rotazione a sinistra decimale. Usato nei calcoli tipo decimale codificato in binario: trascurare.
RR		Come RL ma rotazione a destra.
RRA		Come RLA, ma a destra.
RRC		Come RLC, ma a destra.
RRCA		Come RLCA, ma a destra.
RRD		Come RLD, ma a destra.
RST		Simile a CALL, ma dirige la chiamata esclusivamente a uno degli indirizzi 0, 8, 10, 18, 20, 28, 30 e 38 (tutti in esadecimale). Questi sono tutti entro la ROM dello Spectrum: vedi in proposito i libri di Ian Logan citati in bibliografia. RST 0 equivale alla reinizializzazione iniziale, al momento in cui si accende (o riaccende) la macchina.
SBC	Pag.	Sottrai, tenendo conto del flag del Carry. Il risultato va in A o in HL.
SCF	Pag.	Posiziona (a 1) ("*set*") il flag del Carry.
SET	Pag.	Posiziona a 1 un dato bit di un certo registro.
SLA	Pag.	Spostamento a sinistra aritmetico. Tutti i bit si muovono di un posto verso sinistra; il bit numero 0 va a 0.
SRA	Pag.	Spostamento a destra aritmetico. Tutti i bit si spostano verso destra di un posto; il bit numero 7 viene copiato entro il bit numero 6 *nonché* nel bit numero 7.
SRL	Pag.	Spostamento verso destra logico. Sposta i bit di un posto verso destra; il bit numero 7 viene posto a 0.
SUB	Pag.	Sottrai, senza tener conto del flag del Carry. Il risultato va in A. (Non esiste un'istruzione SUB HL,r: se ve ne serva una, ponete il flag del Carry a 0 e usate SBC).
XDR	Pag.	OR esclusivo fra i bit di eguale posto in A e nel registro indicato. Il risultato va in A.

5 Modifiche dei flag Zero e Carry

Questa appendice illustra quali istruzioni modificano il valore dei flag Zero e Carry, ed in che modo. I simboli usati hanno il seguente significato:

- \- nessuna modifica (inalterato)
- * dipende dal risultato dell'operazione
- 1 posto ad 1 indipendentemente dal risultato
- 0 posto a 0 indipendentemente dal risultato
- \+ posto ad 1 se A =(HL), a 0 in tutti gli altri casi.

Istruzione	C	Z	Istruzione	C	Z	Istruzione	C	Z
ADC	*	*	IND	–	*	RET	–	–
ADD (8-bit)	*	*	INDR	–	1	RETI	–	–
ADD (16-bit)	*	–	INI	–	*	RETN	–	–
AND	0	*	INIR	–	1	RL	*	*
BIT	–	*	JP	–	–	RLA	*	–
CALL	*	*	JR	–	–	RLC	*	*
CCF	*	–	LD A, I	–	*	RLCA	*	–
CP	*	*	LD A, R	–	*	RLD	–	*
CPD	–	+	LD (all others)	–	–	RR	*	*
CPDR	–	+	LDD	–	–	RRA	*	–
CPI	–	+	LDDR	–	–	RRC	*	*
CPIR	–	+	LDI	–	–	RRCA	*	–
CPL	–	–	LDIR	–	–	RRD	–	*
DAA	*	*	NEG	*	*	RST	–	–
DEC	–	–	NOP	–	–	SBC	*	*
DI	–	–	OR	0	*	SCF	1	–
DJNZ	–	–	OTDR	–	1	SET	–	–
EI	–	–	OTIR	–	1	SLA	*	*
EX	–	–	OUT	–	–	SRA	*	*
EXX	–	–	OUTD	–	*	SRL	*	*
HALT	–	–	OUTI	–	*	SUB	*	*
IM	–	–	POP	–	–	XOR	0	*
IN A (n)	–	–	PUSH	–	–			
IN r, (C)	–	*	RES	–	–			

6 Codici operativi dello Z 80

Questa è una lista completa dei codici operativi (opcode) dello Z 80, in ordine alfabetico di sigla mnemonica. In questo elenco, n sta a indicare un numero di 1 byte; nn un numero di 2 byte; e d il valore a 1 byte dello spostamento (offset), scritto in complemento a 2. Si osservi come tutti i numeri di 2 byte abbiano il byte basso *premesso* a quello alto. Esempi:

LD BC, nn ha per opcode 01 nn: così LD BC,732F dà 01 2F 73
LD A,(iY+d) ha per opcode FD 7E d: così LD A,(IY+07) dà FD 7E 07

Questa tabella degli opcode si basa su quella pubblicata dalla Zilog Inc. (costruttrice dello Z 80). Un corrispondente elenco di opcode ordinati per codice numerico è riportata nel Manuale Sinclair, nell'appendice A: da notare che ivi le sigle mnemoniche sono riportate in lettere minuscole.

ADC A, (HL)	8E
ADC A, (IX + d)	DD8Ed
ADC A, (IY + d)	FD8Ed
ADC A, A	8F
ADC A, B	88
ADC A, C	89
ADC A, D	8A
ADC A, E	8B
ADC A, H	8C
ADC A, L	8D
ADC A, n	CEn
ADC HL, BC	ED4A
ADC HL, DE	ED5A
ADC HL, HL	ED6A
ADC HL, SP	ED7A
ADD A, (HL)	86
ADD A, (IX + d)	DD86d
ADD A, (IY + d)	FD86d
ADD A, A	87
ADD A, B	80
ADD A, C	81
ADD A,D	82
ADD A, E	83
ADD A, H	84
ADD A, L	85
ADD A, n	C6n
ADD HL, BC	09
ADD HL, DE	19
ADD HL, HL	29
ADD HL, SP	39
ADD IX, BC	DD09
ADD IX, DE	DD19
ADD IX, IX	DD29
ADD IX, SP	DD39
ADD IY, BC	FD09
ADD IY, DE	FD19
ADD IY, IY	FD29
ADD IY, SP	FD39
AND (HL)	A6
AND (IX + d)	DDA6d
AND (IY + d)	FDA6d
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND n	E6n
BIT 0, (HL)	CB46
BIT 0, (IX + d)	DDCBd46
BIT 0, (IY + d)	FDCBd46
BIT 0, A	CB47
BIT 0, B	CB40
BIT 0, C	CB41
BIT 0, D	CB42
BIT 0, E	CB43
BIT 0, H	CB44
BIT 0, L	CB45
BIT 1, (HL)	CB4E
BIT 1, (IX + d)	DDCBd4E
BIT 1, (IY + d)	FDCBd4E
BIT 1, A	CB4F
BIT 1, B	CB48
BIT 1, C	CB49
BIT 1, D	CB4A
BIT 1, E	CB4B
BIT 1, H	CB4C
BIT 1, L	CB4D
BIT 2, (HL)	CB56
BIT 2, (IX + d)	DDCBd56
BIT 2, (IY + d)	FDCBd56
BIT 2, A	CB57
BIT 2, B	CB50
BIT 2, C	CB51
BIT 2, D	CB52
BIT 2, E	CB53
BIT 2, H	CB54
BIT 2, L	CB55
BIT 3, (HL)	CB5E
BIT 3, (IX + d)	DDCBd5E
BIT 3, (IY + d)	FDCBd5E
BIT 3, A	CB5F
BIT 3, B	CB58
BIT 3, C	CB59
BIT 3, D	CB5A
BIT 3, E	CB5B
BIT 3, H	CB5C
BIT 3, L	CB5D
BIT 4, (HL)	CB66
BIT 4, (IX + d)	DDCBd66
BIT 4, (IY + d)	FDCBd66
BIT 4, A	CB67
BIT 4, B	CB60
BIT 4, C	CB61
BIT 4, D	CB62
BIT 4, E	CB63
BIT 4, H	CB64
BIT 4, L	CB65
BIT 5, (HL)	CB6E
BIT 5, (IX + d)	DDCBd6E
BIT 5, (IY + d)	FDCBd6E
BIT 5, A	CB6F
BIT 5, B	CB68
BIT 5, C	CB69
BIT 5, D	CB6A
BIT 5, E	CB6B
BIT 5, H	CB6C
BIT 5, L	CB6D
BIT 6, (HL)	CB76
BIT 6, (IX + d)	DDCBd76
BIT 6, (IY + d)	FDCBd76
BIT 6, A	CB77
BIT 6, B	CB70
BIT 6, C	CB71
BIT 6, D	CB72
BIT 6, E	CB73
BIT 6, H	CB74
BIT 6, L	CB75
BIT 7, (HL)	CB7E
BIT 7, (IX + d)	DDCBd7E
BIT 7, (IY + d)	FDCBd7E
BIT 7, A	CB7F
BIT 7, B	CB78
BIT 7, C	CB79
BIT 7, D	CB7A
BIT 7, E	CB7B
BIT 7, H	CB7C
BIT 7, L	CB7D
CALL C, nn	DCnn
CALL M, nn	FCnn
CALL NC, nn	D4nn
CALL nn	CDnn
CALL NZ, nn	C4nn
CALL P, nn	F4nn
CALL PE, nn	ECnn
CALL PO, nn	E4nn
CALL Z, nn	CCnn

CCF	3F
CP (HL)	BE
CP (IX + d)	DDBEd
CP (IY + d)	FDBEd
CPA	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP n	FEn
CPD	EDA9
CPDR	EDB9
CPI	EDA1
CPIR	EDB1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX + d)	DD35d
DEC (IY + d)	FD35d
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ, d	10d
EI	FB
EX (SP), HL	E3
EX (SP), IX	DDE3
EX (SP), IY	FDE3
EX AF, AF'	08
EX DE, HL	EB
EXX	D9
HALT	76
IM 0	ED46
IM 1	ED56
IM 2	ED5E
IN A, (C)	ED78
IN A, (n)	DBn
IN B, (C)	ED40
IN C, (C)	ED48
IN D, (C)	ED50
IN E, (C)	ED58
IN H, (C)	ED60
IN L, (C)	ED68
INC (HL)	34
INC (IX + d)	DD34d
INC (IY + d)	FD34d
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD23
INC IY	FD23
INC L	2C
INC SP	33
IND	EDAA
INDR	EDBA
INI	EDA2
INIR	EDB2
JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9
JP C, nn	DAnn
JP M, nn	FAnn
JP NC, nn	D2nn
JP nn	C3nn
JP NZ, nn	C2nn
JP P, nn	F2nn
JP PE, nn	EAnn
JP PO, nn	E2nn
JP Z, nn	CAnn
JR C, d	38d
JR, d	18d
JR NC, d	30d
JR NZ, d	20d
JR Z, d	28d
LD (BC), A	02
LD (DE), A	12
LD (HL), A	77
LD (HL), B	70
LD (HL), C	71
LD (HL), D	72
LD (HL), E	73
LD (HL), H	74
LD (HL), L	75
LD (HL), n	36n
LD (IX + d), A	DD77d
LD (IX + d), B	DD70d
LD (IX + d), C	DD71d
LD (IX + d), D	DD72d
LD (IX + d), E	DD73d
LD (IX + d), H	DD74d
LD (IX + d), L	DD75d
LD (IX + d), n	DD36dn
LD (IY + d), A	FD77d
LD (IY + d), B	FD70d
LD (IY + d), C	FD71d
LD (IY + d), D	FD72d
LD (IY + d), E	FD73d
LD (IY + d), H	FD74d
LD (IY + d), L	FD75d
LD (IY + d), n	FD36dn
LD (nn), A	32nn
LD (nn), BC	ED43nn
LD (nn), DE	ED53nn
LD (nn), HL	22nn
LD (nn), IX	DD22nn
LD (nn), IY	FD22nn
LD (nn), SP	ED73nn
LD A, (BC)	0A
LD A, (DE)	1A
LD A, (HL)	7E
LD A, (IX + d)	DD7Ed
LD A, (IY + d)	FD7Ed
LD A, (nn)	3Ann
LD A, A	7F
LD A, B	78
LD A, C	79
LD A, D	7A
LD A, E	7B
LD A, H	7C
LD A, I	ED57
LD A, L	7D
LD A, n	3En
LD B, (HL)	46
LD B, (IX + d)	DD46d
LD B, (IY + d)	FD46d
LD B, A	47
LD B, B	40
LD B, C	41
LD B, D	42
LD B, E	43
LD B, H	44
LD B, L	45
LD B, n	06n
LD BC, (nn)	ED4Bnn
LD BC, nn	01nn
LD C, (HL)	4E
LD C, (IX + d)	DD4Ed
LD C, (IY + d)	FD4Ed
LD C, A	4F
LD C, B	48
LD C, C	49
LD C, D	4A
LD C, E	4B
LD C, H	4C
LD C, L	4D
LD C, n	0En
LD D, (HL)	56
LD D, (IX + d)	DD56d
LD D, (IY + d)	FD56d
LD D, A	57
LD D, B	50
LD D, C	51
LD D, D	52
LD D, E	53
LD D, H	54
LD D, L	55
LD D, n	16n
LD DE, (nn)	ED5Bnn
LD DE, nn	11nn
LD E, (HL)	5E
LD E, (IX + d)	DD5Ed
LD E, (IY + d)	FD5Ed
LD E, A	5F
LD E, B	58
LD E, C	59
LD E, D	5A
LD E, E	5B
LD E, H	5C
LD E, L	5D
LD E, n	1En
LD H, (HL)	66
LD H, (IX + d)	DD66d
LD H, (IY + d)	FD66d
LD H, A	67
LD H, B	60
LD H, C	61
LD H, D	62
LD H, E	63
LD H, H	64
LD H, L	65
LD H, n	26n
LD HL, (nn)	2Ann
LD HL, nn	21nn
LD I, A	ED47
LD IX, (nn)	DD2Ann
LD IX, nn	DD21nn
LD IY, (nn)	FD2Ann
LD IY, nn	FD21nn
LD L, (HL)	6E
LD L, (IX + d)	DD6Ed
LD L, (IY + d)	FD6Ed
LD L, A	6F
LD L, B	68
LD L, C	69
LD L, D	6A
LD L, E	6B
LD L, H	6C
LD L, L	6D
LD L, n	2En
LD SP, (nn)	ED7Bnn
LD SP, HL	F9
LD SP, IX	DDF9
LD SP, IY	FDF9
LD SP, nn	31nn
LDD	EDA8
LDDR	EDB8
LDI	EDA0
LDIR	EDB0
NEG	ED44
NOP	00
OR (HL)	B6
OR (IX + d)	DDB6d
OR (IY + d)	FDB6d
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6n
OTDR	EDBB
OTIR	EDB3

OUT (C), A	ED79
OUT (C), B	ED41
OUT (C), C	ED49
OUT (C), D	ED51
OUT (C), E	ED59
OUT (C), H	ED61
OUT (C), L	ED69
OUT (n), A	D3n
OUTD	EDAB
OUTI	EDA3
POPAF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
RES 0, (HL)	CB86
RES 0, (IX + d)	DDCBd86
RES 0, (IY + d)	FDCBd86
RES 0, A	CB87
RES 0, B	CB80
RES 0, C	CB81
RES 0, D	CB92
RES 0, E	CB83
RES 0, H	CB84
RES 0, L	CB85
RES 1, (HL)	CB8E
RES 1, (IX + d)	DDCBd8E
RES 1, (IY + d)	FDCBd8E
RES 1, A	CB8F
RES 1, B	CB88
RES 1, C	CB89
RES 1, D	CB8A
RES 1, E	CB8B
RES 1, H	CB8C
RES 1, L	CB8D
RES 2, (HL)	CB96
RES 2, (IX + d)	DDCBd96
RES 2, (IY + d)	FDCBd96
RES 2, A	CB97
RES 2, B	CB90
RES 2, C	CB91
RES 2, D	CB92
RES 2, E	CB93
RES 2, H	CB94
RES 2, L	CB95
RES 3, (HL)	CB9E
RES 3, (IX + d)	DDCBd9E
RES 3, (IY + d)	FDCBd9E
RES 3, A	CB9F
RES 3, B	CB98
RES 3, C	CB99
RES 3, D	CB9A
RES 3, E	CB9B
RES 3, H	CB9C
RES 3, L	CB9D
RES 4, (HL)	CBA6
RES 4, (IX + d)	DDCBdA6
RES 4, (IY + d)	FDCBdA6
RES 4, A	CBA7
RES 4, B	CBA0
RES 4, C	CBA1
RES 4, D	CBA2
RES 4, E	CBA3
RES 4, H	CBA4
RES 4, L	CBA5
RES 5, (HL)	CBAE
RES 5, (IX + d)	DDCBdAE
RES 5, (IY + d)	FDCBdAE
RES 5, A	CBAF
RES 5, B	CBA8
RES 5, C	CBA9
RES 5, D	CBAA
RES 5, E	CBAB
RES 5, H	CBAC
RES 5, L	CBAD
RES 6, (HL)	CBB6
RES 6, (IX + d)	DDCBdB6
RES 6, (IY + d)	FDCBdB6
RES 6, A	CBB7
RES 6, B	CBB0
RES 6, C	CBB1
RES 6, D	CBB2
RES 6, E	CBB3
RES 6, H	CBB4
RES 6, L	CBB5
RES 7, (HL)	CBBE
RES 7, (IX + d)	DDCBdBE
RES 7, (IY + d)	FDCBdBE
RES 7, A	CBBF
RES 7, B	CBB8
RES 7, C	CBB9
RES 7, D	CBBA
RES 7, E	CBBB
RES 7, H	CBBC
RES 7, L	CBBD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET PO	E0
RET Z	C8
RETI	ED4D
RETN	ED45
RL (HL)	CB16
RL (IX + d)	DDCBd16
RL (IY + d)	FDCBd16
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RLA	17
RLC (HL)	CB06
RLC (IX + d)	DDCBd06
RLC (IY + d)	FDCBd06
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLCA	07
RLD	ED6F
RR (HL)	CB1E
RR (IX + d)	DDCBd1E
RR (IY + d)	FDCBd1E
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RRA	1F
RRC (HL)	CB0E
RRC (IX + d)	DDCBd0E
RRC (IY + d)	FDCBd0E
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC A	0F
RRD	ED67
RST 0	C7
RST 10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8	CF
SBC A, (HL)	9E
SBC A, (IX + d)	DD9Ed
SBC A, (IY + d)	FD9Ed
SBC A, A	9F
SBC A, B	98
SBC A, C	99
SBC A, D	9A
SBC A, E	9B
SBC A, H	9C
SBC A, L	9D
SBC A, n	DEn
SBC HL, BC	ED42
SBC HL, DE	ED52
SBC HL, HL	ED62
SBC HL, SP	ED72
SCF	37
SET 0, (HL)	CBC6
SET 0, (IX + d)	DDCBdC6
SET 0, (IY + d)	FDCBdC6
SET 0, A	CBC7
SET 0, B	CBC0
SET 0, C	CBC1
SET 0, D	CBC2
SET 0, E	CBC3
SET 0, H	CBC4
SET 0, L	CBC5
SET 1, (HL)	CBCE
SET 1, (IX + d)	DDCBdCE
SET 1, (IY + d)	FDCBdCE
SET 1, A	CBCF
SET 1, B	CBC8
SET 1, C	CBC9
SET 1, D	CBCA
SET 1, E	CBCB
SET 1, H	CBCC
SET 1, L	CBCD
SET 2, (HL)	CBD6
SET 2, (IX + d)	DDCBdD6
SET 2, (IY + d)	FDCBdD6
SET 2, A	CBD7
SET 2, B	CBD0
SET 2, C	CBD1
SET 2, D	CBD2
SET 2, E	CBD3
SET 2, H	CBD4
SET 2, L	CBD5
SET 3, (HL)	CBDE
SET 3, (IX + d)	DDCBdDE
SET 3, (IY + d)	FDCBdDE
SET 3, A	CBDF
SET 3, B	CBD8
SET 3, C	CBD9
SET 3, D	CBDA
SET 3, E	CBDB
SET 3, H	CBDC
SET 3, L	CBDD
SET 4, (HL)	CBE6
SET 4,(IX + d)	DDCBdE6
SET 4, (IY + d)	FDCBdE6
SET 4, A	CBE7
SET 4, B	CBE0
SET 4, C	CBE1
SET 4, D	CBE2
SET 4, E	CBE3
SET 4, H	CBE4
SET 4, L	CBE5
SET 5, (HL)	CBEE
SET 5, (IX + d)	DDCBdEE
SET 5, (IY + d)	FDCBdEE
SET 5, A	CBEF
SET 5, B	CBE8
SET 5, C	CBE9
SET 5, D	CBEA

SET 5, E	CBEB
SET 5, H	CBEC
SET 5, L	CBED
SET 6, (HL)	CBF6
SET 6, (IX + d)	DDCBdF6
SET 6, (IY + d)	FDCBdF6
SET 6, A	CBF7
SET 6, B	CBF0
SET 6, C	CBF1
SET 6, D	CBF2
SET 6, E	CBF3
SET 6, H	CBF4
SET 6, L	CBF5
SET 7, (HL)	CBFE
SET 7, (IX + d)	DDCBdFE
SET 7, (IY + d)	FDCBdFE
SET 7, A	CBFF
SET 7, B	CBF8
SET 7, C	CBF9
SET 7, D	CBFA
SET 7, E	CBFB
SET 7, H	CBFC
SET 7, L	CBFD
SLA (HL)	CB26
SLA (IX + d)	DDCBd26
SLA (IY + d)	FDCBd26
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SRA (HL)	CB2E
SRA (IX + d)	DDCBd2E
SRA (IY + d)	FDCBd2E
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRL (HL)	CB3E
SRL (IX + d)	DDCBd3E
SRL (IY + d)	FDCBd3E
SRL A	CB3F
SRL B	CB38
SRL C	CB39
SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SUB (HL)	96
SUB (IX + d)	DD96d
SUB (IY + d)	FD96d
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6n
XOR (HL)	AE
XOR (IX + d)	DDAEd
XOR (IY + d)	FDAEd
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR n	EEn

7 HELPA

Eccovi un versatile programma di utilità, scritto in Basic per consentirvi di modificarlo secondo i vostri gusti in modo facile, e che può essere di grande aiuto per la messa a punto, il caricamento e l'esecuzione dei programmi in linguaggio macchina. La sigla è derivata da "Hex Editor, Loader and Partial Assembler". Deriva da uno messo a punto da me per lo ZX 81, ma modificato in modo da tener conto delle migliori caratteristiche dello Spectrum.
1. Impostatelo tramite la tastiera, e poi salvatelo con

SAVE"helpa"LINE 5

una volta che vi siate accertati che non ci sono errori.
2. Date il RUN. Vi verrà richiesto lo spazio di memoria da riservare. Prima comparirà il messaggio Inizio area L.M. = 32000? (ENTER = SI) con relativa richiesta di un INPUT. La pressione di un qualsiasi tasto diverso da ENTER verrà interpretata come indicazione che si desidera un indirizzo di inizio per il linguaggio macchina diverso dal valore standard 32000, che si è utilizzato per tutto il libro, e ne verrà allora richiesto il valore.
Dopo di che viene eseguito un CLEAR, con riposizionamento del valore di RAMTOP, e viene stampato questo nuovo valore nonché l'indirizzo di inizio dell'area del linguaggio macchina, per controllo.
3. Poi viene richiesto il numero di byte per i dati, che precedono l'area del programma in linguaggio macchina. Questi andranno successivamente inseriti con opportuni POKE. (Se credete, potete aggiungere un programmino *ad hoc* in Basic.)
4. Viene stampato anche questo valore, nonché il vero indirizzo iniziale del programma in linguaggio macchina.
5. Viene quindi richiesta l'impostazione del codice in esadecimale, facendo apparire un cursore rosso lampeggiante, in forma di [>].
6. A questo punto potete introdurre (e successivamente, usando idonei comandi di controllo descritti più sotto, manipolare) il codice oggetto del linguaggio macchina in esadecimale. A mano a mano che il codice viene battuto, il programma provvede a eliminare gli eventuali spazi (potete quindi usarli dove volete, per chiarezza nell'impostazione), a suddividere quanto introdotto in gruppi di due caratteri, e a stamparli sullo schermo su 10 colonne. Alla fine di ciascun gruppo viene aggiunto un doppietto formato da ** come marcatore; e il cursore si sposta alla fine del codice appena impostato.
Per fare un esempio, se provate a impostare in successione

"al□e23□□4"∗ENTER (notare gli spazi), questo verrà visualizzato come

al e2 34 ∗∗x [>]

(la stessa forma avrebbe avuto se aveste impostato ale234 oppure al□e2□34).
7. Ora potete fare seguire un nuovo gruppo di codici esadecimali, che verrà aggiunto automaticamente a partire dalla posizione del cursore. I due ∗∗ che rimangono visibili sono solo per comodità, e vengono omessi quando il codice viene effettivamente caricato nella zona appositamente riservata sopra la RAMTOP, cosicché non è affatto necessario che i vari gruppi corrispondano a distinte istruzioni dello Z 80 (possono essere di qualsiasi lunghezza risulti comoda): tuttavia, può essere utile attenersi a tale formato, che rende più facili le successive verifiche.
8. Inoltre, si possono introdurre apposite istruzioni di "controllo". Per distinguerle, queste devono essere il *primo* carattere di ciascun gruppo (quando occorrono); i caratteri che seguono vengono in genere trascurati, salvo nei pochi casi descritti più oltre.
I comandi di controllo sono:

g	(*Go*)	Lancio dell'esecuzione della routine in linguaggio macchina
l	(*Load*)	Caricamento della routine in linguaggio macchina sopra la RAMTOP
m	(*Move*)	Sposta in avanti il cursore di 1 o più posizioni
n	(*Negative*)	Sposta il cursore indietro di 1 o più posizioni
p	(*Print*)	Stampa sullo schermo il listato esadecimale
r	(*Relative*)	Usato per calcolare automaticamente l'offset dei salti relativi
s	(*Save*)	Salva il programma su cassetta
x	(*eXcise*)	Cancella dal listato uno o più "doppietti".

Facciamo ora seguire una descrizione più dettagliata di questi comandi, in un ordine diverso per maggior comodità.

m, n Il comando ma, dove a è un numero, sposta in avanti il cursore di a spazi; na lo sposta indietro di a spazi. Esiste una protezione contro il superamento dei limiti del listato. Se si omette **a**, viene assunto uguale a 1.

x Un comando xa, dove **a** è un numero ≥ 0, cancella i successivi **a** doppietti di caratteri (inclusi i ∗∗) a partire dal cursore. Il listato visualizzato non viene modificato sino a quando non si preme **p**. Se si omette **a**, esso viene assunto uguale a 1.

Per *aggiungere* del testo, posizionare il cursore immediatamente prima del punto dove si vuole l'inserimento, e iniziare l'impostazione dei nuovi caratteri. Tutto quello che segue verrà spostato in avanti per fare posto, ma solo il nuovo testo verrà visualizzato fino a che non si preme p.

p Stampa (sullo schermo) il testo attuale, e sposta il cursore in cima.

s Salva su cassetta il programma, incluso il listato esadecimale attuale. Un'opzione permette di assegnare un nome qualsiasi al programma salvato. Per lanciare un programma salvato, dare il SAVE *prima* di usare 1 e g; quindi caricare (LOAD) il Basic, fare GO TO 200 (*non* RUN!!), e solo poi l e g.

l Carica il linguaggio macchina, privato dei ** superflui, nell'area sopra la RAMTOP fissata in precedenza; provvede ad aggiungere il RET finale.

g Lancia il programma. Poi il controllo torna a HELPA (a scanso di un blocco totale!).

r Una comodità per rendere facile il calcolo dei salti *relativi* tanto noiosi da calcolare "a mano", perché, oltre a controllare l'entità dello spostamento, occorre convertirne in esadecimale il valore in complemento a 2; eppure tanto comodi per permettere una facile "rilocabilità" del linguaggio macchina. Il funzionamento è circa questo:

(a) Al momento dell'impostazione del codice in linguaggio macchina, porre eguali a 00 tutti i valori dei salti relativi (offset).

(b) Per inserire il valore corretto per un dato salto, spostare il cursore immediatamente prima dello 00 e cancellarlo con x. Ora impostare **rn**, dove **n** è la posizione (in notazione decimale) della *destinazione* del salto, che si può ricavare dalla lista che compare sullo schermo nel seguente modo. Si assegnino numeri d'ordine alle righe e alle colonne del listato esadecimale partendo da 0, così:

```
     0 1 2 3 4 5 6 7 8 9

0
1
2
3
.
.
.
.
```

prendere quindi **n** = xy (**x** numero di riga, **y** numero di colonna). (Per esempio, per il byte posto nella riga 17, colonna 5, si farà r175). Il fatto che ci siano 10 colonne rende facile individuare il relativo numero d'ordine. (Potete, se lo desiderate, modificare il programma in modo che il cursore venga spostato sul punto di destinazione, e da qui calcolare il valore del salto: tuttavia questo sistema risulta più lento, a causa dei molti spostamenti di cursore che dovete effettuare.)
(c) Ora premete p per visualizzare il listato revisionato, che include il valore corretto del salto.
(d) Posizionare il cursore prima del successivo 00 di un altro salto relativo, e ripetere l'intero procedimento.
(e) Osservate come il programma tenga conto automaticamente (trascurandoli) dei vari ** sparsi qua e là, e come fornisca direttamente il complemento a 2 richiesto: se il salto risultasse superare i limiti ammessi, lo segnalerà.
(f) Il tutto può sembrare complicato, pertanto eccovi un esempio, che utilizza la routine per lo scrolling di una colonna del capitolo 14. Impostate successivamente i gruppi di codice esadecimale; il risultato sarà:

Nuova RAMTOP:	31999
Inizio area linguaggio macchina	32000
Dati	da 32000 a 31999
Chiamata linguaggio macchina	USR 32000

CODICE ESADECIMALE:

```
01 20 00 ** dd 21 1f 58 ** 3e
15 ** dd 56 20 ** dd 72 00 **
dd 09 ** 3d ** b8 ** 20 00 **
```

dove il doppietto 00 sottolineato corrisponde al valore del salto che vogliamo calcolare. (Le indicazioni per l'area dati possono apparire un po' strane: se volete, modificate il programma in modo che venga segnalato: Dati: nessuno quando il numero di byte dati vale 0).
Usando n spostate il cursore immediatamente prima del doppietto citato, così:

```
..........
dd 09 ** 3d ** b8 **x20[>]00 **
```

e poi premete x per cancellare lo 00. In questo caso l'istruzione è JRNZ

loop, come sappiamo; e loop è in corrispondenza al codice dd della seconda linea di codice esadecimale, che si trova nella riga 1, colonna 2 (rammentate che si parte da 0 in entrambi i casi); di conseguenza dovete dare il comando r12. Eseguite.

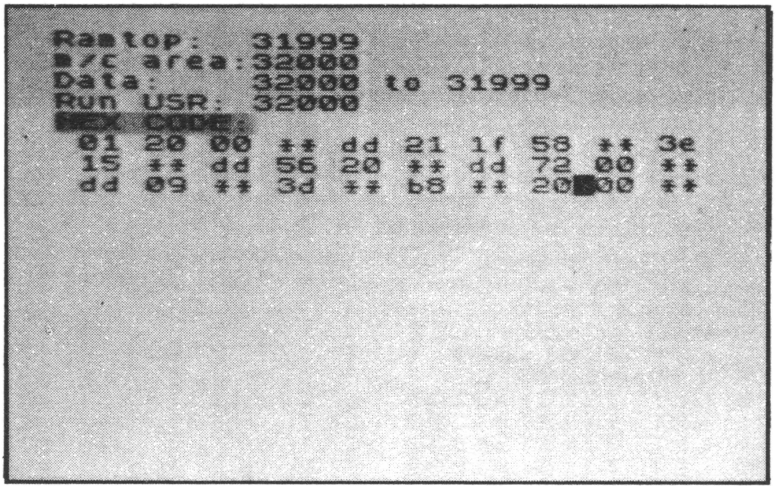


FIG. A7.1. Come appare lo schermo prima che HELPA proceda a calcolare il valore di un salto relativo

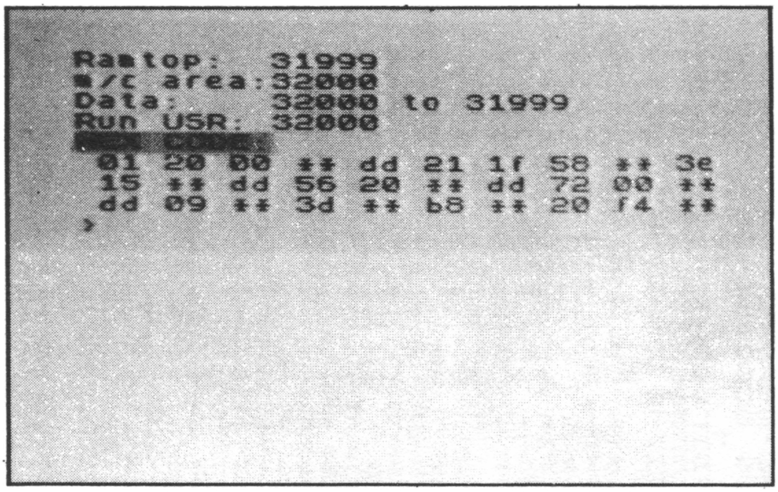


FIG. A7.2. È stato inserito il corretto valore f4 del salto

Dopo qualche istante, lo schermo si cancella, poi viene stampato un nuovo listato in cui lo 00 è sostituito da f4, il corretto valore del salto relativo.

Ed eccovi il listato completo del programma HELPA:

```
   1 REM HELPA ©1982 Ian Stewart
 & Robin Jones
   5 POKE 23609,50
  10 LET rt=32000: INPUT "Inizio
 area L/M =32000?"'"(ENTER = SI)
 ";a$: IF a$<>"" THEN INPUT "Ini
zio zona L/M? ";rt
  40 CLEAR (rt-1): LET rt=PEEK 2
3730+256*PEEK 23731+1
  60 CLS : PRINT "Nuova RAMTOP",
rt-1'"Inizio area L/M",rt
  80 INPUT "Numero di  byte dati
 ? ";d
 100 PRINT "Dati da ";rt;" a ";r
t+d-1
 110 PRINT "Chiamata L/M","USR "
;rt+d'
 120 PRINT PAPER 6;"Codice HEX:"
 130 LET ci=0: LET h$="": GO SUB
 400
 150 DIM f(20): LET f(1)=700
 160 LET f(6)=800: LET f(7)=900:
 LET f(8)=1000
 170 LET f(10)=1100: LET f(11)=1
200: LET f(12)=1300: LET f(13)=1
400: LET f(14)=1500
 180 LET f(16)=1600
 200 INPUT i$
 210 LET a=0
 220 LET a=a+1
 230 IF a>LEN i$ THEN GO TO 300
 240 IF i$(a)<>" " THEN GO TO 22
0
 250 LET i$=i$( TO a-1)+i$(a+1 T
O ): GO TO 230
 300 IF CODE i$(1)>=103 THEN GO
TO 500
 310: LET i$=i$+"**": LET h$=h$(
 TO 2*ci)+i$+h$(2*ci+1 TO )
 330 GO SUB 450: GO SUB 600: GO
SUB 400: GO TO 200
 400 REM Cursore
 410 PRINT AT 5+INT (ci/10),3*(c
i-10*INT (ci/10)); FLASH 1; INK
2;">";: RETURN
 450 REM Cancella cursore
 460 PRINT AT 5+INT (ci/10),3*(c
i-10*INT (ci/10));" ";: RETURN
 500 REM Routine tastiera
 510 GO SUB f(CODE i$(1)-102): G
O TO 200
 520 LET ci=ci+1: IF ci=10*r(ci/
10) THEN PRINT "  ";
 600 REM Stampa testo aggiunto
 610 FOR j=1 TO LEN i$/2: PRINT
i$(2*j-1 TO 2*j)+" ";
 620 LET ci=ci+1: IF ci=10*INT (
ci/10) THEN PRINT "  ";
 630 NEXT j: RETURN
 700 REM Lancio routine LM
 710 CLS : LET y=USR (rt+d): RET
URN
 800 REM Caricam. oltre RAMTOP
```

```
 810 LET h$=h$+"c9": LET j=rt+d-
1: LET i=-1
 820 LET j=j+1
 830 LET i=i+2
 840 IF i>LEN h$ THEN RETURN
 850 IF h$(i)="*" THEN GO TO 830
 860 POKE j,16*(CODE h$(i)-48-39
*(h$(i)>"9"))+CODE h$(i+1)-48-39
*(h$(i+1)>"9")
 870 GO TO 820
 900 REM Sposta avanti cursore
 910 GO SUB 450: IF LEN i$=1 THE
N LET cm=1: GO TO 930
 920 LET cm=VAL i$(2 TO )
 930 LET ci=ci+cm: IF ci>LEN h$/
2 THEN LET ci=LEN h$/2
 940 GO SUB 400: RETURN
1000 REM Sposta indietro cursore
1010 GO SUB 450: IF LEN i$=1 THE
N LET cm=1: GO TO 1030
1020 LET cm=VAL i$(2 TO )
1030 LET ci=ci-cm: IF ci<0 THEN
LET ci=0
1040 GO SUB 400: RETURN
1100 REM Stampa
1110 PRINT AT 5,0;
1120 DIM z$(32): FOR r=1 TO 17:
PRINT z$;: NEXT r
1130 PRINT AT 5,0;" ";: LET ci=0
1140 FOR j=1 TO LEN h$/2
1150 PRINT h$(2*j-1 TO 2*j)+" ";
: LET ci=ci+1
1160 IF ci=10*INT (ci/10) THEN P
RINT " ";
1170 NEXT j: GO SUB 400: RETURN
1300 REM Salti relativi
1310 LET jci=VAL i$(2 TO ): LET
js=jci-ci-1: GO SUB 2000
1320 IF js>=-128 AND js<=127 THE
N GO TO 1340
1330 INPUT "Valore non corretto:
 premi ENTER";a$: RETURN
1340 IF js<0 THEN LET js=js+256
1350 LET x1=INT (js/16): LET x0=
js-16*x1
1360 LET x$=CHR$ (x1+48+39*(x1>9
))+CHR$ (x0+48+39*(x0>9))
1370 LET h$=h$( TO 2*ci)+x$+h$(2
*ci+1 TO )
1380 LET ci=ci+1
1390 GO SUB 1100: RETURN
1400 REM SAVE
1410 INPUT "Nome del file? ";n$:
 IF n$="" THEN LET n$="helpa"
1420 POKE rt+d+LEN h$,201
1430 SAVE n$CODE rt,d+LEN h$+1
1440 PRINT #1;"Per ricaricare:"'
"LOAD""";n$;""" CODE ": PAUSE 20
0: RETURN
1600 REM Cancella
1610 IF LEN i$=1 THEN LET k=1: G
O TO 1630
1620 LET k=VAL i$(2 TO )
1630 LET h$=h$( TO 2*ci)+h$(2*ci
+2*k+1 TO ): RETURN
2000 REM Aggiusta asterischi
```

```
2010 IF js<0 THEN LET w$=h$(2*jc
i+1 TO 2*ci): GO TO 2030
2020 LET w$=h$(2*ci+1 TO 2*jci)
2030 LET sc=0
2040 FOR t=1 TO LEN w$: IF w$(t)
="*" THEN LET sc=sc+1
2050 NEXT t
2060 IF js<0 THEN LET js=js+sc/2
: GO TO 2080
2070 LET js=js-sc/2
2080 RETURN
```

Finito di stampare il settembre 1984
presso Lito Velox - Trento
Printed in Italy

A scuola con il PET/CBM
Trenta programmi didattici per ogni tipo di PET/CBM matematica, fisica, statistica, ecc.
160 pagine, 13.000 lire, sigla PDC 15

Il manuale dell'Atom
L'edizione italiana del manuale originale di questo personal computer, ora importato in Italia.
300 pagine, 18.500 lire, sigla PDC 16

Il libro del Commodore VIC 20
Questo libro è inteso come supplemento al manuale della macchina, e presenta numerose caratteristiche del VIC.
156 pagine, 12.000 lire, sigla PDC 17

Il debug nei personal computer
Il "debug", in informatica, è la messa a punto dei programmi, la ricerca e la correzione finale degli errori. Si tratta di una operazione di grande importanza, che si può effettuare in vari modi, tutti trattati in questo libro.
144 pagine, 15.000 lire, sigla PDC 18

Programmazione in Basic per l'uomo d'affari
Questo libro è per gli uomini d'affari che vogliono imparare ad usare il calcolatore e parte dalla convinzione che sia più facile per un uomo d'affari imparare a programmare, che per un programmatore imparare la gestione di una azienda
256 pagine, 19.000 lire, sigla PDC 19

Imparate il Basic con lo ZX-81
È il microcomputer più venduto nel mondo. Ma come ogni computer, necessita di software e documentazione: questo libro contiene 37 programmi che illustrano tutte le caratteristiche e le capacità della macchina.
132 pagine, 13.000 lire, sigla PDC 20

Dal Basic al Pascal
Questa eccezionale guida rende facile per chiunque passare dalla propria conoscenza del Basic ad una perfetta padronanza del Pascal, un linguaggio a 2 livelli che richiede uno spazio inferiore di memoria nel vostro computer.
88 pagine, 10.000 lire, sigla PDC 21

Imparate il Basic con il Texas TI 99/4A
Il Texas TI 99/4A vi può aiutare nell'apprendimento delle lingue, della matematica, tenere la contabilità della vostra famiglia. Basta conoscere il linguaggio giusto: questo libro ve lo insegna.
264 pagine, 22.000 lire, sigla PDC 22

A scuola con il Texas TI 99/4A
Scritto da un insegnante, questo è un libro di software per la scuola. Gli argomenti sono presentati in ordine di difficoltà crescente e sono tutti trattati negli studi medi inferiori e superiori. Ma non mancano applicazioni varie e giochi.
212 pagine, 18.000 lire, sigla PDC 23

Come usare il Commodore 64
Il lettore troverà nel libro non solo una facile guida per imparare il Basic del Commodore 64, ma anche una fonte completa di informazioni sull'installazione, gli accessori, gli user's group ed il software disponibile.
140 pagine, 18.000 lire, sigla PDC 24

Imparate il Basic con lo Spectrum
Questo libro è stato scritto per chi non ha alcuna esperienza di computer, sia per chi è già esperto ma necessita di maggiori informazioni di quelle date dal manuale. Partendo dai principi di base, il lettore può arrivare alle più avanzate tecniche utilizzabili.
192 pagine, 19.000 lire, sigla PDC 25

A scuola con il Commodore 64
Scritto da due insegnanti, questo è un libro di software per la scuola. I programmi sono in Basic, sono tra i pochi programmi originali, studiati e scritti in Italia e non tradotti.
160 pagine, 17.000 lire, sigla PDC 26

Imparate il Basic con l'IBM Personal Computer
I 53 capitoli e le 5 appendici, vivacizzati da sorridenti vignette, porteranno in breve anche coloro che non hanno esperienza del linguaggio dell'IBM ad apprezzare le moltissime capacità di questo personal.
320 pagine, 26.000 lire, sigla PDC 27

Introduzione al Lisp
Questo volume presenta gli elementi esseziali del Lisp in modo piano e accessibile, con uno stile ancor più leggibile dall'impostazione "domanda e risposta" e riporta e commenta una sessantina di programmi e di routines immediatamente utilizzabili.
150 pagine, 16.000 lire, sigla PDC 28

Programmi in Basic per l'elettronica
Una raccolta di routine d'aiuto alla soluzione di problemi che frequentemente si incontrano, soprattutto in fase di progettazione e impegnano moltissimo tempo in calcoli di solito noiosi e ripetitivi.
138 pagine, 14.000 lire, sigla PDC 29

cognome e nome

indirizzo

cap. località

(partita iva o codice fiscale)

firma

il formato della cartolina è conforme alle vigenti norme postali

cedola di commissione libraria

franco muzzio editore

via bonporti, 36

35141 padova

Un libro per tutti coloro che possiedono uno home Computer Sinclair Spectrum e vogliono penetrare più a fondo nella capacità di questa piccola, grande macchina: per esplorare tutte le caratteristiche del sistema operativo dello Spectrum che non sono sfruttabili efficientemente con il Basic, ma solo con il linguaggio macchina, dalle memorie degli attributi e dello schermo alle variabili di sistema.

Sono fornite alcune fra le routine più interessanti, per realizzare divertenti "scorrimenti" sullo schermo, per rinumerare le linee di un programma, per ricerche veloci, per la grafica ad alta velocità.

franco muzzio & c. editore

L. 16.000 (15.686)

ISBN 88-7021-266-1

30

Ian Stewart e Robin Jones

Il linguaggio macchina dello Spectrum